NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

USABILITY ANALYSIS OF THE CHANNEL APPLICATION PROGRAMMING INTERFACE

by

Christopher A. Brown

June 2003

Thesis Advisor:	Geoffrey Xie
Second Reader:	Rudolph P. Darken

13. ABSTRACT ***(maximum 200 words)***

The Channel Application Programming Interface (API) provides a tool for loosely coupling components in Component Based Design (CBD) projects. In the thesis that proposed and developed the API, the author provided a technical analysis of the API's performance with respect to communication metrics. However, only the author/designer has ever used the API; hence, no analysis was accomplished with respect to Usability attributes. The project sponsor desires public release of the API. However, a usability analysis is first required to ensure wide acceptance and use of the API.

In order to analyze the API, an analysis method and associated metrics are required. Little work has been done in the field of Human Computer Interface (HCI) with respect to treating an API as an interface and programmers as the end users. This thesis follows an IEEE published test protocol and well known HCI approaches to test the API for general usability attributes as well as to investigate specific features of the API. Specifically, the analysis will test the API's ability to explain itself during first-time exposure in order to gain acceptance.

The results from testing the API are used to determine necessary enhancements to the API and its documentation.

14. SUBJECT TERMS Channel API; Application Programming Interface; Usability Analysis			**15. NUMBER OF PAGES** 147
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89) Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

USABILITY ANALYSIS OF THE CHANNEL APPLICATION PROGRAMMING INTERFACE

Ensign, United States Navy
B.S. Computer Science, San Diego State University, 2002

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2003

Author: Christopher A. Brown

Approved by: Geoffrey Xie
Thesis Advisor

Rudolph P. Darken
Second Reader

Peter Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The Channel Application Programming Interface (API) provides a tool for loosely coupling components in Component Based Design (CBD) projects. In the thesis that proposed and developed the API, the author provided a technical analysis of the API's performance with respect to communication metrics. However, only the author/designer has ever used the API; hence, no analysis was accomplished with respect to usability attributes. The project sponsor desires public release of the API, especially within the Department of Defense (DoD). However, a usability analysis is first required to ensure wide acceptance and use of the API.

In order to analyze the API, an analysis method and associated metrics are required. Little work has been done in the field of Human Computer Interface (HCI) with respect to treating an API as an interface and programmers as the end users. This thesis follows an IEEE published case-study and well known HCI usability analysis methods to test the API for general usability attributes as well as to investigate specific features of the API. Specifically, the analysis will test the API's ability to explain its functionality during first time exposure. The API's acceptance will depend on its success or failure to convey its purpose quickly during this initial exposure.

The results from testing the API are used to determine required enhancements to the API and its documentation.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to extend my appreciation to Dr. Geoffrey Xie for his help and encouragement during the course of preparing this thesis and in completing the Masters program at Naval Postgraduate School. In addition, I extend my appreciation to Dr. Rudy Darken for his practical introduction to Human Computer Interface Analysis. And to both, appreciation is extended for the guided freedom given in completing the study contained herein.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. CHANNEL API ANALYSIS PROBLEM

An Application Programming Interface (API) is a library of methods that provide programmers with access to predefined programming constructs and operations. An example is the Java Swing API provided for development of a Graphical User Interface (GUI) [Ref 4]; it provides an interface to the complex system operations necessary for GUI design. APIs such as Swing are important tools for programmers in both the civilian and military sectors. They abstract away many low level tasks thereby allowing programmers to focus more on their own specific design goals. In addition, well designed APIs provide reusable units of code that eliminate the need for programmers to recreate the functionality represented by the API. This is perhaps the biggest benefit of a well designed API.

Unfortunately, during the design of many APIs, more thought is given to the end functionality than is given to ensuring the reuse or usability of the API. While common Object Oriented practices and programming discourse rules help to balance functionality with usability, APIs are generally seen as tools which are less important than the end projects to which they are lending their support. Thus, many potentially useful units of code are discarded, thereby defeating the reuse goal of Object Oriented Programming.

A possible two part approach to overcoming this wasteful employment of API design is described here. The first part involves application of User Centered Design (UCD) principles during the development of an API. This includes for example ensuring that class, interface, and method signatures convey there purposes in efficient and easy to learn manners so that the code describes itself. User Centered Design can also be applied in the development of descriptive documentation. The second part is to employ human factors analysis during and after API development. This involves usability analysis on the API to identify where and why an API fails to present itself as a reusable tool.

This thesis is an exercise of the second part, usability analysis, to provide enhancements to the user centered design efforts of the Channel API's designer and, to provide input into the user centered improvements of the API documentation. Specifically, this analysis will focus on the API's ability to explain itself during first-time exposure. It is the position of this paper that success of the API is dependent on its presentation during first-time exposure; if the API and its documentation do not explain its functionality quickly, then the API will likely not gain acceptance.

The Channel API is a well developed and reusable Java Package. It incorporates well known programming principles such as event-based programming and loose-coupling of components; and, it is a very powerful tool for use in Component[1] Based Design (CBD) projects. The designer put much effort into ensuring that the functionality of the API is efficient. In fact, in the thesis that contains the API, there is a thorough technical analysis of the API with respect to communication metrics such as throughput and delay [Ref 1]. However, this analysis did not include any usability testing, which is needed to ensure the API will be utilized if made available to the public.

The API author did consider some UCD principles during its design. For example, the requirement for an API to incorporate meaningful method names, class names, and method signatures[2] was discussed in the thesis [Ref 1]. However, none of these design qualities underwent any usability testing. Additionally, learnability and comprehension of the underlying Channel Model was not tested; and, the learnability impact of abstracting event-based programming within a message like model was not considered or tested.

An immediately noticeable deficiency in the presently available Channel API is its inadequate documentation. The majority of the class, interface, and method comments found in the documentation are not helpful and do not promote usability. For instance, they do not contain any useful descriptions of their purposes or uses. Instead, they only contain one to a few lines of minimally descriptive text, which were most likely more useful to the author during design than to any potential users. Usability analysis will

[1] *Components* in this paper refer to code modules in a single application not to stand-alone processes; and, *Component Based Design* is the development and assembling of these code modules in a single application.

[2] The term *method signature* refers to the return type, name, parameter types, and parameter order of a method. This term is synonymous with the term *function prototype* commonly used in C/C++ discussions.

allow the identification of where better documentation, examples, and explanations are required. In addition, an overview of the Channel API is needed as part of its distribution. This may be in the form of a user's manual, a progressive example, or both.

B. MOTIVATION AND BENEFITS OF THESIS

It is desired by the thesis sponsor to release the Channel API to the public. This does not sound difficult. One needs to simply place the class files on a web-page and perhaps provide some HTML pages as documentation. However as with many publicly available APIs, the difficulty in learning and using the API will likely deter programmers from wanting to use it. Thus, the Channel API will sit on the shelf of a library and never realize the purpose for which it was designed. Therefore, the main intent of this thesis is to identify enhancements to the API that will produce a more usable version that supports rapid learning and promotes reuse.

Another intention is to present an example of the usability analysis of an API. This thesis will employ a published method of API analysis and will use well known methods of HCI usability analysis. This example will contribute to the field of HCI by providing a further case study of usability analysis on an API.

In addition, the results of the usability testing will reveal how programmers can employ user centered design principles when developing APIs. This will include useful commenting of methods to enhance comprehension, application of naming conventions to enhance learnability, and overview documentation requirements to enhance the comprehension of the underlying API model.

C. ORGANIZATION OF THESIS

The remainder of this thesis is organized as follows. Chapter II provides background information on the Channel API and application of HCI usability analysis methods to an API. The latter summarizes the key points of a published API testing protocol followed in this paper. Chapter III provides an overview of the analysis method used in this paper. Chapter IV describes the Channel API analysis with respect to basic

API functionality; this chapter includes the test plan, the test results, and the recommended enhancements for the API. Chapter V, like Chapter IV, describes the analysis of the Channel API, but, this time with respect to advanced functionality of the API. Finally, Chapter VI concludes the thesis discussion and introduces possible follow on work.

II. BACKGROUND INFORMATION

This chapter provides an overview of the Channel API and an introduction to the application of usability analysis to an API.

A. DESCRIPTION OF CHANNEL API

This section is an introduction to the Channel API. This introduction includes an overview of the underlying model employed by the API, a description of the basic classes and interfaces found in the API, a description of its advanced classes and interfaces, and an overview of the suggested use of the API. The information provided here is a result of extensive use of the API in conjunction with examination of documentation and source code.

1. Channel Model

The Channel API is designed around the model of a communication channel which may have multiple talkers and listeners. With this model, the Channel API incorporates two programming concepts. The first is Component Assembly, which involves providing a means of assembling separately developed components into a larger application. For the Channel API, this function is performed with instances of the *Channel* class, which accepts registration by talker and listener components. The second concept is Event-Driven Data Flow, in which an object experiencing an event pushes the event or data off to another object for processing. The Channel API implements this concept through the flow of event objects from talker to channel to listener objects.

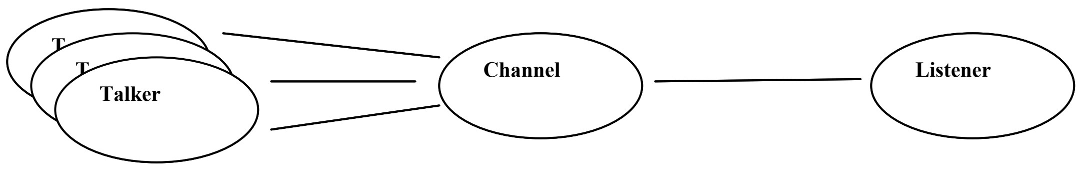
a. Component Assembly

The Channel Model contains three main components. They are talkers, channels, and listeners. Talkers are developed as event producers and listeners are developed as event consumers. Channels are used to assemble or associate talker and listener components. This association occurs when talkers and listeners register with a channel. It is then the channel's responsibility to control all the interaction between its registered talkers and listeners.

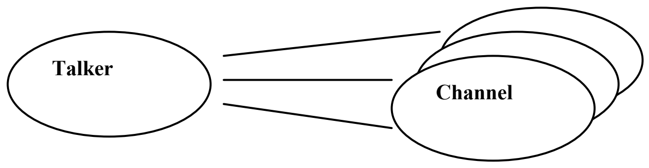
The relationships between the talkers, listeners, and channels in the Channel model may be one to many, many to one, or one to one in nature. Both talkers and listeners may register with many channels or a single channel depending on how the programmer wishes to assemble them within an application. Conversely, a channel may have one or many talkers and listeners. In addition, a channel may itself become a listener on another channel, thereby allowing the piping of events from one channel system to another. All of these relationships may change during program execution in any manner desired.

By viewing the channel as a connection point between talkers and listeners, these same types of relationships may be expressed between talkers and listeners. For example, a talker may push events to a channel that in turn delivers these events to many listeners; or, many talkers may push events to a channel that delivers them to a single listener; these numerous possible configurations give programmers great flexibility in how they assemble components in an application. In addition, the channel removes the necessity for the talkers and listeners to be tightly coupled[3]. Some possible component configurations are shown in Figure 1 below.

[3] *Tight Coupling* describes the relationship between two objects which have strong dependencies on each other either physical with references to one another, operational with reliance on operations of one another, or both.



Many to One
Many Talkers to one Channel and Many Talkers to one Listener



One to Many
One Talker to many Channels/Listeners

Figure 1. Possible Channel Model Component Configurations

b. Event-Driven Data Flow

Event-Driven programming is a well known and easy to follow concept. In it, a producer object creates events and pushes them off to a consumer object. The consumer then performs any required processing of the event. In the Channel Model, talkers perform the role of event producer and listeners perform the role of event consumer. The channel acts as a kind of event switchboard that allows delivery of events to multiple consumers, reception of events from multiple producers, and event scheduling.

Once a pair of talker and listener components is assembled, the talker, as event producer, initiates all interactions when it experiences and pushes an event to a channel. The channel then determines which of its listeners requires delivery of the event and delivers it to them. Finally, the listener, as event consumer, receives the event from the channel and performs the required processing. This simplex flow of events can occur in conjunction with any of the described component configurations already discussed.

Figure 2 below depicts the flow of an event, e, from talker to channel to listener in a simple one to one component structure.

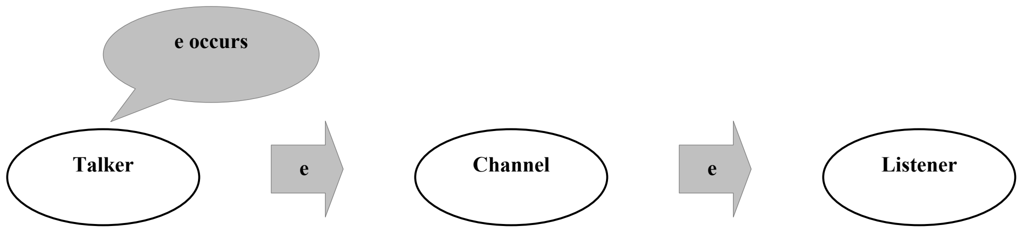


Figure 2. Channel Model Event Flow

Although the events flow in a simplex fashion, duplex communication may be achieved due to the API's extensibility. Since any Java Object can fulfill the role of talker, the listener objects can function as talkers on the same channel or on a different channel for the purpose of sending events back to the original talker, which would itself implement the *ChannelListener* interface.

The combination of this event-driven flow of application specific data objects with the flexible component assembly mechanism allows the functionalities of the model's producers and consumers to remain completely separate. The talkers may produce and pre-process the events in any way required by the application and continue on in its purpose after pushing the events off to one or more channels. Likewise, the listeners may consume and process events without any concern or knowledge of the talker's purpose or existence. This makes the Channel API a powerful tool for support of component based design projects.

2. Basic Objects

The basic objects used in the Channel API follow directly from the description of the Channel Model. There is a *Channel* class that is instantiated to produce channel objects; there is a *ChannelListener* interface that must be implemented by any user defined class that is to perform the role of listener; there are talker objects, which may be any Java Object; and, there are event objects which may be any Java Object or an instance of the pre-defined *ChannelEvent* class. The following sub-sections describe each of these basic objects.

a. Channel Class

The *Channel* class is the central class used for the basic functionality of the Channel API. Its main purpose is to assemble talker and listener components. Its main functions are talker and listener management and event reception and delivery. The *Channel* class fulfills these functions through the use of both built in and customizable classes and interfaces.

The assembling of talker and listener components is achieved through the use of channel specific IDs and various add methods. Every channel in an application is assigned a unique integer ID. This ID is set when the channel is instantiated. The ID allows programmers to identify channels for specific purposes, and is used when a talker or listener is added to a channel. This static ID allows programmers to associate multiple talkers and multiple listeners with a channel without knowing having to know the dynamic physical reference to the channel object.

Additionally, talkers and listeners are registered with a channel through the use of multiple add methods. These add methods allow programmers to set priorities and filters when a talker or listener is registered or to let the channel set these attributes as defaults. The *Channel* class methods used to assemble talkers and listeners as well as those that allow programmers access to them, are contained in Table 1 below.

Method Name	**Arguments**	**Description**
Channel	*int channel_ID*	Constructor that creates a Channel with a specific ID.
Channel	*int channel_ID*, *ChannelScheduler*	Constructor that creates a Channel with ID and custom Scheduler.
addListener	*ChannelListener*	Used to add a single Listener to a Channel.
addListener	*ChannelListener*, *int priority*	Adds a Listener with a priority.
addListener	*ChannelListener, ChannelFilter*	Adds a Listener with a filter.
addListener	*ChannelListener, ChannelFilter, int priority*	Adds a Listener with a filter and a priority.
addListener	*ChannelListenerItem*	Adds a Listener encapsulated in a ChannelListenerItem.
addTalker	*java.lang.Object*	Adds any java Object as a Talker.
addTalker	*java.lang.Object, int priority*	Adds a Talker with an assigned priority.

Table 1. Channel Methods Used for Component Assembly

Talker and listener management is done internally by the *Channel* class. A channel object keeps an internal listing of registered talkers and listeners. These listings are available to programmers through the use of get methods, which return Java

Vectors containing all the registered talkers or listeners. In addition, the *Channel* class provides methods for testing if either of the listings is empty. For individual talkers or listeners, the class provides methods to check if the object is registered with the Channel.

Method Name	**Arguments**	**Return Type**
getListeners	None	*java.util.Vector*
getTalkers	None	*java.util.Vector*
hasListeners	None	*boolean*
hasTalkers	None	*boolean*
isRegisteredListener	*ChannelListener*	*boolean*
isRegisteredTalker	*java.lang.Object*	*boolean*
removeListener	*ChannelListener*	*boolean*
removeTalker	*java.lang.Object*	*boolean*

Table 2. Channel Methods for Talker/Listener Management

Event reception is straight forward and involves only basic objects. A channel receives an event when a registered talker invokes one of the channel's talk methods. These talk methods allow a talker to push events to a channel in a variety of ways. A Talker may push a pre-defined *ChannelEvent* object, a generic Java Object, or a generic Java Object with a specified priority. If the pushed event is a generic Java Object, then the Channel encapsulates it in a *ChannelEvent*. These talk methods are listed in Table 3 below.

Method Name	**Arguments**	**Description**
talk	*ChannelEvent*	Delivers a *ChannelEvent* object to the channel.
talk	*java.lang.Object talker, java.lang.Object event*	Delivers the talker object and an event that is a generic Java Object.
talk	*java.lang.Object talker, java.lang.Object event, int priority*	Delivers the talker and event Object in addition to an event priority.

Table 3. Channel Class Methods Used For Event Reception

Event delivery involves both basic and advanced objects of the Channel API, and is customizable using advanced features. The advanced features are discussed in depth in later sub-sections, and are mentioned only briefly here. A channel uses an internal dispatcher thread to deliver the events pushed to it from all its talkers. In its simplest form, delivery is accomplished when a channel's dispatcher thread calls a listener's *receiveEvent* method using a default scheduler object. This happens inside the channel and is not accessible to programmers. However, the Channel API offers programmers two interfaces to customize the event delivery process.

The first interfac interface is the *ChannelScheduler* interface. Implementation of this interface allows programmers to define how events are scheduled for delivery. Programmers may define any scheduling algorithm desired and may base scheduling on any event attribute desired. This customized scheduler is set when a channel object is instantiated and replaces the default FIFO Scheduler.

The second interface is the *ChannelFilter* interface. Implementation of this interface allows programmers to specify which events are delivered by a channel to a specific listener. A filter object is associated with a specific listener object; this association may occur when a listener is added to the channel or afterwards with a call to the channel's *addFilter* method. A filter may also be removed, thereby stopping the filtering process for the associated listener.

The channel also provides a self-dispatching mechanism for listeners. This feature allows the channel to assign a thread to a specific listener for event delivery. Events are then delivered by this thread instead of by the channel's dispatcher. This is an advanced feature and is discussed more in depth in the section on advanced objects.

b. Channel Listener Interface

Components fulfill the role of listener through implementation of a simple Java interface, *ChannelListener*. This interface requires implementation of a single method, *receiveEvent(ChannelEvent event)*. The method is invoked by a channel object when delivering an event to a registered listener. The channel delivers the event inside of a *ChannelEvent* object, which may be unpacked by the listener.

As simple as the interface is, there is a synchronization issue that must be considered. Since a listener may register with multiple channels, it is possible that more than one channel may call the listener's *receiveEvent* method simultaneously. In order to avoid this, the method should be synchronized. This will ensure that only one channel invokes the method at a time.

Method Name	**Arguments**	**Description**
receiveEvent	*ChannelEvent*	Called by the channel to which a listener is registered to deliver an event to the listener.

Table 4. Channel Listener Interface Methods

c. Talker Objects

Any Java Object may fulfill the role of talker. There is no interface to implement. However, a talker must be registered with a channel in order to talk on it. This is accomplished with the channel's *addTalker* methods. Also, talker components push events to a channel using the channel's talk methods. The event pushed may either be a Java Object or an instance of *ChannelEvent*.

d. Encapsulation Classes

There are two encapsulation classes used in the Channel API. The first is the *ChannelEvent* class, which is used by the talker, channel, and listener components. The second is the *ChannelListenerItem*, which is used by the channel and listener components.

The *ChannelEvent* class is used to encapsulate events. The attributes of a channel event are talker, event, and event priority. The talker and event may both be generic Java Objects and are both required to create a *ChannelEvent*. The event priority is an integer value assigned to the event; it is assigned a default value when it is not provided at creation. However, a method is provided to set the priority when desired.

The *ChannelEvent* is used by three API components. The channel uses the *ChannelEvent* to encapsulate all events when pushed to it by a registered talker; and, the channel uses it to deliver events to registered listeners. A talker may use the *ChannelEvent* to encapsulate the events it pushes to a channel; and, a listener processes it upon receiving it from the channel. The *ChannelEvent* class provides a number of methods for setting and getting access to its data members and for accessing attributes associated with its creation and handling. Table 5 below contains these methods and their uses.

Method Name	Arguments/Returns	Description
getEvent	Returns: *java.lang.Object*	Provides access to the encapsulated event object; used by channels for filtering and by listeners for processing of received events.
getEventClass	Returns: *java.lang.Class*	Provides access to the event Class; used by channels for filtering and by listeners for processing of received event.
setEventPriority	Takes: *int*	Sets the priority for the encapsulated event object; used by talker when packaging an event for pushing to channel; and, used by channel when it receives an un-encapsulated event.
getEventPriority	Returns: *int*	Returns the event's priority; used by channel for scheduling and listener for processing.
getTalker	Returns: *java.lang.Object*	Provides access to the encapsulated talker object; used by channel for filtering and the listener for

		processing of received events.
getTalkerClass	Returns: *java.lang.Class*	Provides access to the encapsulated talker's Class; used by the channel for filtering and the listener for processing of events.
getTalkerPriority	Returns: *int*	Provides access to the talker's priority; used by channel for scheduling.
setTalkerPriority	Takes: *int*	Sets talker priority; used by talker and channel.
getTimeStamp	Returns: *long*	Provides access to the system time as set by channel at reception of an event.

Table 5. *ChannelEvent* Class Methods

The *ChannelListenerItem* is used to encapsulate information about a listener. This information includes the *ChannelListener* object itself, its *ChannelFilter* objects, and its priority. The *Channel* class uses this container class to store its associated listeners and information about them. This storage takes place when a listener registers with the channel. The listener may be registered pre-encapsulated in a *ChannelListenerItem* or un-encapsulated; either way, the channel will store the listener in this encapsulation class.

This class is also involved in the self-dispatching mechanism for listeners. This feature allows events to be delivered to a listener using a dedicated thread instead of the channel's own dispatching thread. Self-dispatching is started, stopped, suspended, or resumed for a listener with calls to *Channel* class methods. When self-dispatching is started, the *ChannelListenerItem* acts as the dedicated thread; it allocates a queue into which the *Channel* object can place incoming events. This is possible due to the class' implementation of the Java *Runnable* interface. When events arrive to the channel destined for a listener, the channel places them into the queue rather than delivering them with the *receiveEvent* method defined in the *ChannelListener* implementation. The purpose for self-dispatching is to avoid delays which may occur when a listener takes an excessive amount of time to receive an event.

Method Name	**Arguments/Returns**	**Description**
startListenerselfDispatch	Takes: *ChannelListener,* Returns: *boolean*	Used to start self-dispatching for a listener.
startListenerselfDispatch	Takes: *ChannelListener, int queueLength* Returns: *boolean*	Used to start self-dispatching for a listener with a customized queue length.
suspendListenerselfDispatch	Takes: *ChannelListener*	Used to temporarily suspend self-dispatching for a registered listener.
resumeListenerselfDispatch	Takes: *ChannelListener*	Used to resume self-dispatching for a registered listener.
stopListenerselfDispatch	Takes: *ChannelListener*	Stops self-dispatching for a registered listener.

Table 6. *Channel Methods* Used for Self-dispatching

3. Advanced Objects

This sub-section describes the advanced classes and interfaces contained in the Channel API. They provide mechanisms for programmers to manage channels, define scheduling routines for event delivery, and to define filters for listener reception of events. Each of these mechanisms is introduced with its corresponding class or interface.

a. ChannelManager Class

The *ChannelManager* class is provided as a predefined and centralized tool for managing channels in an application. This involves overriding all the *Channel* class methods with an additional channel ID parameter, which allows reference to a channel using the ID alone. Thus, the registering of talker and listener components with a specific channel and talking on a specific channel is mediated by the channel manager.

Use of the channel manager is a more efficient means of using the Channel API than the means available using only basic objects. Instead of the programmer writing code to create and track all the channels in an application, the channel manager does the work already. This feature is intended for large applications containing many channels.

One subtle feature, which is to be the focus of the testing in this thesis, is the way in which the channel manager creates channels. Currently, this happens when the first call to add a talker or a listener is made with a specific ID. At this call, the channel manager checks to see if a channel with the specified ID exists yet. If it does then the add method executes the same as in the channel class; if the channel does not exist, then it is created with the specified ID.

As seen in the constructors shown in Table 7, a *ChannelManagerAuthority* may be provided to the channel manager when it is created. When present, the manager uses the authority to enforce access rules for channels. The *ChannelManagerAuthority* is an interface and is described in the next sub-section.

The following table contains a sample of the *ChannelManager* class methods which over-ride the *Channel* class methods.

Method Name	**Arguments**	**Description**
ChannelManager	None	Constructor that creates a channel manager.
ChannelManager	*ChannelManagerAuthority*	Constructor that creates a channel with an authority object to provide access control.
addListener	*ChannelListener,* *int channel_ID*	Used to add a single listener to a channel.
addListener	*ChannelListener*, *int priority,* *int channel_ID*	Adds a listener with a priority.
addListener	*ChannelListener,* *ChannelFilter,* *int channel_ID*	Adds a listener with a filter.
addListener	*ChannelListener,* *ChannelFilter,* *int priority,* *int channel_ID*	Adds a listener with a filter and a priority.
addListener	*ChannelListenerItem,* *int channel_ID*	Adds a Listener encapsulated in a *ChannelListenerItem*.
addTalker	*java.lang.Object,* *int channel_ID*	Adds any java Object as a talker.
addTalker	*java.lang.Object, int priority,*	Adds a talker with an assigned

	int channel_ID	priority.
talk	*ChannelEvent,* *int channel_ID*	Delivers a ChannelEvent object to the Channel.
talk	*java.lang.Object talker,* *java.lang.Object event,* *int channel_ID*	Delivers the talker object and an event that is a generic Java Object.
talk	*java.lang.Object talker,* *java.lang.Object event,* *int priority,* *int channel_ID*	Delivers the talker and event Object in addition to an event priority.

Table 7. Important *ChannelManager* Class Methods Overriding *Channel* Class Methods

b. ChannelManagerAuthority Interface

This interface is a troublesome feature of the API and is to receive a lot of attention in the usability analysis contained in later chapters of this thesis. The intent of the interface as explained by the designer in his thesis is for an implementation of the authority interface to control and organize all channels and its participants in a centralized manner [Ref 1]. However, the methods required in the interface imply an access control role instead of an organizing role. In addition, the method names and arguments do not appear to be capable of providing the level of control implied.

The five required methods are as follows:

- *boolean isTalkerAuthorized(java.lang.Object talker , int channel_id)*
- *boolean isListenerAuthorized(ChannelListener listener, int channel_id)*
- *int getTalkerPriority(java.lang.String talkerClassName)*
- *int getListenerPriority(java.lang.String listenerClassName)*
- *ChannelScheduler getSchedulerForChannel(int channel_id)*

The first two methods seem to function well for access control since their arguments target a specific talker or listener on a specific channel. However, the second two ask for a priority for a talker or listener but their arguments do not allow specification of a specific talker or listener and no way of targeting a specific channel. In addition, it is not apparent why a channel's scheduler would be required for access control.

It is assumed that the function of this interface is access control; however, this is not apparent from the thesis' description and does not seem efficiently possible with the method arguments. Thus, discussion of the interface's purpose and required enhancements are left until after the usability analysis is completed.

c. ChannelFilter Interface

This simple interface allows for an implementing class to define a filtering mechanism for screening events prior to their delivery to a listener. A filter is associated with a specific listener when it registers with a channel or later with a call to an add method. A list of filters for each listener is stored with the filter inside a *ChannelListenerItem* by the channel object to which a listener is registered. When an event arrives to the channel, it polls each of its listeners to determine which of them to deliver the event to. During this poll, if a listener has one or more filters, then the filter is applied to determine if the event gets delivered to the listener or not; and, if no filters are present then all events are delivered to the listener.

A predefined filter class is provided in the Channel API in the form of the *ChannelEventFilter* Class. It demonstrates the filtering mechanism and gives examples of the various class attributes that may be used for filtering.

d. ChannelScheduler Interface

This interface allows the programmer to define how a *Channel* object will schedule the delivery of events. When an event arrives to a channel, the channel encapsulates the event in a *ChannelEven*t object prior to pushing it to its scheduler. The channel's dispatcher then pulls events from the scheduler, leaving it to the scheduler to define the event delivery order.

Every channel has a scheduler; it is either a custom defined class, which implements the *ChannelScheduler* interface, or, it is a default *FIFOScheduler*. A custom scheduler is set for a channel as a parameter to the *Channel* constructor; and, a default is set when no such parameter is provided. The Channel API provides three pre-defined implementations of the *ChannelScheduler* interface; they are, the *FIFOScheduler*, used as default, the *PerTalker_RR_Scheduler*, and the *PriorityScheduler.*

The two required methods for this interface are:

- *void push(ChannelEvent event)*
- *ChannelEvent pull()*

4. Overview of Use

This section provides an overview of the suggested use of the Channel API. The use is explained with respect to using the API in a component based design project. When using the Channel API in such a project, application design involves development of three types of components; the first type is the talker components, which will produce event objects; the second is the listener components, which will received and process the event objects; and, the third is the main application that will assemble the first two types of components using a channel manager and channels.

a. ID Schema

The communication within the Channel Model is designed to eliminate the necessity of talker and listener components having physical reference to a channel object. This is possible through the use of channel IDs. When a talker or listener interacts with a channel via a channel manager, they do so using only the channel ID. This allows for the association between talkers and listeners with a channel to be dynamic; and, eliminates the passing of channel references between components.

Thus, the first consideration when using the Channel API should be definition of a channel ID schema. This schema will define unique channel IDs that identify which channels talker and listener components will use. The schema is part of the overall project specification and provides a communication contract between developers of the talker/listener components and the developer of the assembling application. For example, in a network simulation application all network flows of a certain type could be assigned to Channel 5; and, all nodes that produce this type of flow would be talkers that push events to Channel 5 and, all nodes that receive this type of flow would be listeners on Channel 5.

b. Channel Management

The next consideration is how the application will manage channels. The intent in the design of the Channel API is for a *ChannelManager* object to mediate access to channels; however, there is also the option to create and manage channels without the use of a channel manager and static channel IDs in a small application. This is a cumbersome task in a large application as it adds to the responsibilities the of the application designer since he or she must then track and manage all channels and their associated talkers and listeners.

Thus, the recommended usage is for the assembling application to use the *ChannelManager* class to assemble all talker and listener components. This provides a predefined and centralized object to manage the channels. If it is desired, single channel objects may be created and used for specific tasks.

c. Talker-Listener Contracts

The Channel API allows loose coupling of talker and listener components. They do not have to wait on each other during execution; and, they do not require reference to each other's methods to coordinate in an application. However, talkers will be responsible to produce specific types of event objects and to deliver them to specific channels; and, listeners will be required to receive and process specific types of events from specific channels. Thus, each talker and listener component should have a contract specified to ensure that the overall application will function as desired when the components are assembled.

For talker components, the contract includes definition of the types of events it will produce, how the events will be packaged, and which channels to deliver each event type to. From the discussion in the section on basic objects, it is recalled that the talker may push any generic Java Object as an event. This event may be pushed directly to the appropriate channels or, it may be encapsulated in a channel event object. For listener components, the contract includes definition of the types of events it will process, an explanation of how it should process the events, and which channels it will receive the events from.

In addition, a listener may be designated as talker on specific channels and, a talker may be designated as a listener on certain channels; thus, a list of channel IDs and there their use for talking or listening is required in both contracts. A suggested contract is given in the Table 7 below.

Channel ID	**Component Role**
1	Talker
2	Talker
3	Listener
4	Listener

Listener Contract		
Event Type Received	**Processing Required**	**Channel Received From**
AirContactData Class Instance	Extract event from *ChannelEvent* and process coordinates.	3

SurfaceContactData Class Instance	Extract event from *ChannelEvent* and process coordinates.	4
Talker Contract		
Event Type Produced	**Pre-Processing**	**Channel Pushed To**
AirContactData Class Instance	Encapsulate in a *ChannelEvent* object with priority set to 5.	1
SurfaceContactData Class Instance	Encapuslate in a *ChannelEvent* object; if CPA < 1,000 yards set priority to 4 else let it default to 0.	2

Table 8. Talker/Listener Component Contract

d. Component Design and Assembly

Once the channel ID schema and the talker/listener contracts are defined, the next step is to design the talker and listener components to fulfill the contracts. This is application specific and done in any way desired by the programmer responsible for each component. The contracts and schema ensure that the components will fit into the application as desired.

Since the component contracts are specified ahead of implementation, the programmer responsible for implementation of the main application can create and assemble the necessary talker and listener components while they are themselves being implemented; this is a benefit of component based design. The Channel API provides the interface for the components to the overall application and for the interface to the components for the assembling application. Thus, the API specific work required in the main application is as follows:

- Create required talkers and listeners as instances of the classes that implement the component contracts.
- Create a channel manager if desired.
- Create the channels within the channel manager at calls its add methods or create them as needed.
- Register the talkers and listeners with the appropriate channels with calls to the *ChannelManager* or *Channel* class add methods.
- Define the logic to support the interaction between the talker/listener components and the channel manager or individual channels.

The last bullet is accomplished in one of two ways. The main application may control all interaction by calling the talk methods; or, the talker components may be implemented as threads that have reference to the channel or channel manager and push the events with talk method calls on these references. The second is the suggested use as it provides the highest amount of independence between the talker components and the main application class.

B. USABILITY ANALYSIS OF AN API

This Section provides an introduction to usability analysis and its application to an API. It includes sub-sections on usability analysis, difficulties of analyzing an API, and a summary of an analysis performed on an API.

1. Usability Analysis Overview

This section is an introduction to the concept of usability analysis. It closely follows the discussion presented by Jeffrey Rubin in his *Handbook of Usability Testing* [Ref 3] where he explains how to plan, design, and conduct effective usability tests; there, he discusses analysis of a product; here, his ideas are discussed with respect to analysis of an interface. This section is not an exhaustive definition of usability analysis. For complete coverage of the subject, one should see the Rubin text or take a course covering Human Computer Interfaces.

Usability analysis is an attempt to measure the usability of an interface based on observation of human interaction with the interface. Testing methods seek to quantify various aspects of the human-interface interaction in ways that can be examined to provide improvements to the interface design. Usability testing differs from classical scientific testing in that it seeks to determine deficiencies and fixes for them whereas traditional research seeks to prove or disprove a hypothesis [Ref 3].

The commonly measured interface attributes are defined in the list below.

- <u>Learnability</u>

  This is a measure of how easy an interface is to learn and how rapidly a user can become productive with the interface.

- <u>Efficiency</u>

  This is a measure of how efficient the interface is when used to perform the task it supports.

- <u>Memorability</u>

  This is a measure of the interface's support for user recall of its functionality.

- <u>Errors</u>

  This is a measure of error rate experienced by users during use of the interface and the interface's support for user recovery from the errors.

- <u>Satisfaction</u>

  This is a measure of user's subjective perception and acceptance of the interface.

In order to effectively measure these attributes, a controlled, planned, and scientific test method is required. Such a method consists of the following elements [Ref 3]:

- Specific problem statements or test objectives that define aspects of interface to be tested.
- Participants representing the intended end users.
- Representation of the actual work environment.
- Observation of user interaction with the interface.
- Collection of quantitative and qualitative performance and preference data[4].
- Recommendations for improvements.

Using these elements, a test method is developed to analyze the interface. The method may be designed for exploration, assessment, validation, or comparison of the interface [Ref 3], with each method type occurring at a certain point in the interface design and having specific objectives. For example, an exploration test may happen early in the design process to test usability of design concepts; an assessment test may happen midway through design to assess findings of the exploratory test and/or to test low-level functionality; a validation test may happen late in design process to verify how the interface or product measures against a defined set of usability standards; and, the comparison test may happen at any time in design process to compare the product or its design to make a comparison between two alternative designs or products.

Regardless of what type of test method is used, Rubin proposes six stages for conducting a usability test [Ref 3]. These are listed and discussed in the following six sub-sections.

a. Test Plan Development

Development of the test plan is the first and most important stage of the test. This plan provides details of all subsequent stages of the test and encompasses all the elements required in a usability test. Its purpose is to communicate the how, when, where, why, who, and what of the test to all parties involved in the interface design; and,

[4] Performance data are measures such as task timing and task accuracy whereas preference data are measures of participant opinion such as responses to subjective questions.

it provides a rigorous guide to keep testing on track [Ref 3]. Typical sections of a test plan include:

- Discussion of Test Purpose
- Listing of Objectives
- Discussion of User Profile
- Discussion of Test Method and Design
- Discussion of Task List
- Discussion of Test Environment
- Discussion of Monitor Role
- Discussion of Measurable Data
- Discussion of Intended Output Report

b. Participant Selection and Acquisition

This stage of the usability test involves definition of the interface user profile and acquisition of participants who match the profile. These activities include characterizing the experience and abilities of intended users, categorization of these abilities and experience levels, choosing the number of participants to test, competency level of participants, participant compensation, and determination of where to find participants. In addition, a means of screening participants is developed using the profile characteristics.

Much thought must be given to the interface's end users to ensure that the right people are tested. Participants must possess the required knowledge level to interact with the interface; they must possess the education level required to perform the tasks for which the interface is designed; in short, they must be a true sampling of the intended interface users. To ensure this happens, a means of screening participants must be developed using the profile characteristics. For example, potential participants may be asked to complete a questionnaire or to answer questions during an interview.

c. Preparation of Test Materials

This stage of the testing involves preparation of materials used to screen participants, collect data, and collect data from participants. Typical test materials and their uses are:

- Participant Screening Questionnaire: used to ensure participants match the user profile.
- Session Script: used to ensure that the test is delivered to every participant in the identical manner.
- Data Collection Forms: used to record both performance and preference data.
- Consent Forms and Non-Disclosure Agreements: used for legal purposes to gain participant permission for session recording and to protect proprietary product information.
- Task Scenarios: used to present the task list contained in the test plan to the participants; these are tasks representing interface features embedded in scenarios that duplicate actual use cases.
- Debriefing Guidelines: used to give structure to the debriefing session, which ensures that the subjective data can be quantified in some manner.

d. Conducting Test

This stage of the usability test is the delivery of the test method discussed in the test plan in a scripted step-by-step manner. The steps involved are test specific; however, they must be delivered in the same way for every participant. Rubin suggests using a series of three checklists to guide test delivery [Ref 3]. The first checklist is applied approximately two weeks before testing begins and is used to shake out any bugs in the test delivery; it requires that the monitor take the test, deliver the test to an initial test subject, correct any errors in the test materials, and secure the necessary equipment and environment. The second checklist is applied the day before testing. It prompts the monitor to setup the equipment, assemble materials, and confirm scheduled participants. The final checklist is applied on the day of the test. It is a step-by-step list of events for the test session from greeting the participant to debriefing and dismissal.

e. Participant Debriefing

This stage of the usability testing is the last opportunity to gather information about a participant's confusion during interaction with the interface. It allows pointed questions with respect to unexplained mistakes; insight is gained through interrogation of participants and review of their actions during testing [Ref 3]. To accomplish this, the monitor must review the participant's responses as recorded on collection forms and ask him/her why a particular error was made.

For effective debriefing, Rubin suggests the following guidelines [Ref 3]:

- Avoid placing participant on guard with criticism.
- Do not react to a participant's answers.
- Review data collection forms and post-test questionnaires.
- Begin by letting participant say whatever comes to mind.
- Begin questions from high-level issues.
- Move to specific issues found on collection forms and questionnaires.
- Focus on understanding the problems and difficulties experienced by participant rather than on solving the problem.

f. Transformation of Data into Recommendations

This final stage involves reviewing the data to draw conclusions about interface features and making recommendations to fix design flaws. This process requires compiling and summarizing data, analyzing data, developing recommendations and reporting recommendations [Ref 3].

Compilation of data involves organizing the data into a form that can be analyzed for each task. Quantitative data can be placed into tables and charts. Hand recorded responses can be transcribed into digital format and grouped. The goal is to have all the data represented in a way that reveals patterns of behavior [Ref 3]. This compilation can occur at the end of each test, end of each day, or at end of testing; however, compiling data in-line with test sessions is advantageous since it allows the monitor to confirm that the data matches the test objectives and to take advantage of the freshness of the test in the monitor's mind.

Summarization of data is task oriented; it is a statistical analysis of both performance and preference data for each task. Statistics for performance data are computed using measurements of time and errors. For task timings, the mean, median, range, and standard deviation of completion times are used to gain understanding of participant performance [Ref 3]; and, errors and completion criteria are used to calculate task accuracy statistics. Statistics for preference data are computed from questions, comments, and debriefing sessions. For example, the number of positive or negative responses to a subjective question about an interface is summed to measure user satisfaction.

Analysis of data is also task oriented. Its purpose is to ensure that the users can perform the tasks using the interface; it is accomplished using the following activities [Ref 3]:

- Identifying the tasks which did not meet pre-defined criterion; these non-criterion tasks will indicate where to focus the analysis.
- Identifying the errors or difficulties causing the tasks to fail.
- Conducting a source of error analysis to determine what part of the interface caused the error.
- Prioritizing the problems by criticality to allow efficient translation of the problems into solutions.

Rubin discusses the development and reporting of recommendations as two distinct activities. For developing recommendations, he suggests focusing on solutions which will have the most impact on the interface design, avoiding political motivations, providing short and long term solutions, indicating areas that need further investigation, and thorough discussion of all problem areas [Ref 3]. The intent is to address the critical problems affecting the interface and to make practical suggestions to their solutions.

Once the recommendations are finalized, a report should be written to present the test and its recommendations. The report should cover three aspects; first, it should explain why and how the test was prepared; second, it should explain what happened during the test; and third, it should present the recommendations [Ref 3]. The first of these is the test plan with addition of any modifications; the second is a report on the performance and preference data summaries made; and, the third is a discussion of the recommendations.

2. Difficulties of API Analysis

An API is by definition an interface. It therefore represents a user's point of interaction with a product; and, like any other interface, it is difficult to measure its usability due to the difficulty in measuring why people make mistakes and what they are thinking when they make them. However, an API suffers from additional difficulties. Since it provides an interface to an abstraction of a concept, it has no physical representation to interact with aside from documentation; and, user interaction with it is less tangible than with other types of interfaces.

The only means of representing an API is with examples and documentation. These representations provide a layer of distraction between the user and the interface and may interfere with analysis of the API if they are not well prepared. Examples are a good representation of the API as they demonstrate the API in action and provide a means of analyzing comprehension of its functionality and the ability of its methods and classes to describe that functionality; however, examples are more of a tool for teaching rather than a means of measuring the API's usability since they do not involve actual use of the API. Since the documentation contains the class and method names of the API and are involved in the API's use, they are necessarily the best interface representation to be used for usability analysis; however, this requires distinguishing between mistakes caused by the documents and those caused by the API; and, it requires including analysis of the documentation as part of the testing.

In addition to the physical representation problem, the measurements available for API analysis are more indirect than those available for other types of interfaces. With a GUI, HCI specialists quickly define time limits for the discovery of a button as a measure of the buttons effective positioning in a window. Likewise, they may count the number of mouse clicks performed prior to clicking a button as an indication of the same attribute. With an API, the measurements are more subtle. They involve measures of a user's mental actions and are dependent on the user's willingness to provide mention of them. Thus, they require defining indications of the mental actions and forcing a user to discuss his or her interaction in a way that measures those indications.

3. Case Study of an API Analysis Using a Think Out Loud Protocol

In an article entitled "Building More Usable APIs" that appeared in the IEEE Software Journal, McLellan et al presented an analysis method that utilizes a think out loud protocol to test usability attributes of an API [Ref 2]. They used the protocol to capture subject interactions with the interface. The interface was represented as a "contrived example" containing approximately 75 percent of the API functionality. The subjects' comments and actions were video taped for later review.

Their subjects were application programmers from the API's target user group. Specifically, the test targeted first time users who were familiar with the C and C++ programming languages, were familiar with general data exchange concepts, and were overall experienced programmers [Ref 2]. Some of the subjects were familiar with the specific data standards used in the API, while some were not; but, they made no distinction between these two groups.

During the test sessions, subjects were asked to examine the contrived example displayed in a text editor. They were encouraged to explain what they would need to know about the API in order to produce the example themselves; and, they were allowed to ask questions to the examiner whenever they became confused [Ref 2]. In addition, at the end of the test sessions subjects were asked to speculate on what other features the API would perform based on the functionality represented in the example; and, they were asked to complete a questionnaire indicating satisfaction and perception.

The test designers had determined through prior research that commented examples could aide rapid learning of the API and could provide a means of measuring usability. Their definition of usability required measurement of the following API attributes [Ref 2]:

- Ease of API Learning
- Efficiency of API Use for Specific Tasks
- Memorability of the API Method Calls
- Misconceptions or Errors Made by Programmers While Using the API
- Programmers' Perceptions of the API

This study reached conclusions with respect to effective use of code examples to describe API functionality and usability of the API and its documentation [Ref 2]. Their conclusions on where and how to use examples were based on the end of test questionnaire and subject comments. In the questionnaire, all of the subjects were able to correctly describe additional API features. They contributed this success to their use of pseudocode to quickly describe the code example's use of API features and to help programmers develop hypotheses about how the API functioned. The code example also supported learning of the API's purpose, its usage protocols, and its usage context [Ref 2].

To reach conclusions on the API's usability, they looked for patterns during review of the taped sessions. These patterns included questions raised by three or more participants, long time spans spent on specific code segments, and number of questions related to a code segment or method call [Ref 2]; these patterns gave indication of which portions of the API would require explanation to support efficient use. Specifically, they found that programmers wanted self descriptive code and, that violation of discourse rules indicated necessary redesigns or documented explanations for the violations.

The benefit of their methodology is that it tests an API during development. This allows usability issues to be addressed during design rather than accounted for in documentation; and, it provides input for the development of reference documentation.

THIS PAGE INTENTIONALLY LEFT BLANK

III. API ANALYSIS METHOD

The analysis method used for this project combines the think-out-loud protocol discussed in Chapter II with a well known usability test development method for producing organized test plans. The protocol used here is based on the one described by McLellan et al [Ref 2]; the procedure discussed in their paper is used as a basis for developing the structure of the testing protocol and for determining suitable attributes to be measured. The usability test development techniques are based on those found in Rubin and summarized in Chapter II [Ref 3]; his techniques are used to structure the tests into a well formed process for collecting the subject data and analyzing the output.

The combination of the two components into an analysis method is done for practical reasons. The McLellan paper is one of few examples of a usability study conducted on an API. Most interface analyses are done on GUIs and other traditional interaction devices. It is therefore hard to find examples of API analysis. The test development procedure described by Rubin is used for its clear and concise methods of preparing, conducting, and reviewing usability tests. Thus, the combination of the two into one analyses method is a natural choice based on available sources of reliable procedures.

A. TEST PROTOCOL

The test protocol used for analysis of the Channel API is derived from the think-out-loud protocol discussed in Section B of the previous chapter. As in the tests performed there, programmers are exposed to the API's functionality during a video taped test session and their responses are used to gather usability information about the API. However, instead of presenting programmers with a contrived example the test in this paper presents them with a sample programming project.

The sample project requires programmers to complete a small component based design project using the Channel API. The project encapsulates a series of tasks that are designed to focus the programmers' attentions on the tested functionality of the API. Its intent is to capture their first time exposure to the API. They do not actually write code for the project; instead, they verbalize their thought process while considering the project. Use of the sample project instead of a contrived example allows observance of actual use of the API rather than examination of an example of its use. This allows collection of data representing the programmers' actual responses as first-time users of the API. These data include comments indicating perception, comprehension, and recognition of appropriate method calls, which indicate effective use of self-descriptive code during API design. It also allows recording of the thought process the programmers follow during actual learning of the API's functionality.

In addition to the sample project, a series of end-of-test questions are given. These require the programmers to respond without reference to the API documentation. The questions are designed to test memorability of method, class, and interface names and learnability of the API's basic functionality and use. For example, a particular question asks programmers to identify from a list the object which initiates the interaction required to deliver an event. Their responses measure learnability and comprehension of the API's event-driven data flow model and the role of each class in that model. A complete listing of the end-of-test questions and their purposes are contained in Section B.2 of Chapter IV.

Together, the project, questions, and exit interviews provide the opportunity for programmers to verbally express their thought process during and after use of the Channel API. These verbal responses are recorded and later analyzed against predefined criteria and hypotheses for each task. The results from this analysis answer the questions defined by the test objectives, which are discussed in the context of the two test plans described in the next section.

B. TEST PLAN DEVELOPMENT

It was desired to test a large portion of the Channel API's functionality. In order to avoid long test sessions and to reduce the complexity of the test, two test-sessions were planned with each focusing on specific API functionality. For each session, a detailed test plan was developed using the procedure given in Rubin's *Handbook of Usability Testing* [Ref 3]. These test plans are presented in their entirety in Section A of Chapters IV and V; an overview of their main components is presented in this section.

For each test plan, a list of objectives was developed indicating the specific aspects of the Channel API to be tested. These objectives are in the form of questions to be answered by observing subjects' interactions with the API. For example, one objective in the first test is to determine if method names and signatures convey their purpose in the Channel Model; and, in the second test, an objective is to determine the perceived purpose of the *ChannelManagerAuthority*. These objectives are used to focus each test on specific areas of interest.

Once the objectives were finalized, a task list was developed which prompts subjects to investigate the features targeted by the objectives. Each task embodies a portion of the test objectives. For clarity, each task is accompanied by a list of goals, a list of test criteria, and a list of hypotheses. The goals represent one or more of the objectives to be met by the task. The test criteria indicate actions to be demonstrated by subjects during task completion. And, the hypotheses are statements of the expected outcomes for each task.

The test criteria play an important part in the collection of subject data. They provide a means of measuring the subject's ability to complete each task using the API; and, they provide a means of specifying the API features that must be understood for task completion to be successful. To develop the test criteria, consideration was given to the interaction that would result between the user and the API during task completion. The methods, classes, interfaces, procedures, and documents that would be encountered were analyzed to determine how they would be used to convey the API's intended use to first-time users quickly and efficiently. These API attributes were then compared to the test objectives and those that gave insight to the objectives were worded as test criteria.

The hypotheses are used to develop focal points to aide in comparing expected problem areas to actual test results for the purpose of quickly identifying required enhancements. For example, one hypothesis predicts that a class comment will cause confusion and suggests an enhancement to remove the confusion. Once the testing is complete, the outcome of the task is compared with the hypothesis; and, if the outcome is as predicted, then the proposed enhancement is available for immediate consideration.

The first test-session and its test plan are designed to analyze the basic functionality of the Channel API. This includes the Channel Model, component assembly mechanisms, and the event-driven data flow model. The task list is comprised of three tasks, one to expose subjects to the role of the talker, one to expose them to the listener role, and one to expose them to component assembly using the *Channel* class. The objectives and test criteria are presented in Sections A.2 and A.5 of Chapter IV. This first test plan also makes use of the end-of-test questions discussed in the Protocol Section of this chapter.

The second test plan is designed to analyze advanced API features provided for channel management and filtering. Its task list is comprised of four tasks. The first three expose subjects to channel management using the *ChannelManager* class, channel management using the *ChannelManagerAuthority* interface, and channel management using both the manager and the authority jointly. The final task exposes subjects to the API's filtering mechanism. The objectives and test criteria are presented in Sections A.2 and A.5 of Chapter V. This test does not make use of the end-of-test questions.

C. SAMPLE PROJECT

The tasks contained in the test plans are embedded in a simple component based design project. This project asks programmers to develop a shipboard contact tracking application. In the first test, they are instructed to develop a *SensorInput* class and an *InputProcessor* class. The first is to act as a talker component and the second is to perform the role of listener. They are given a brief description of the Channel API's talker, channel, and listener objects and told to use the API to assemble their two classes. For the second test, the project is a continuation of the first. Subjects are instructed to redesign the tracking application to make use of various Channel API features provided for management of multiple channels and for filtering.

Each project divides its tasks into smaller sub-tasks, with each representing one or more of the test criteria listed in the test plans. The project as presented to the participants is located in Appendix A in the sections entitled Test 1 Project Write-Up and Test 2 Project Write-Up.

The goal of the sample project was to present subjects with an actual use scenario requiring them to examine the available representation of the interface. This required them to learn both the underlying model used by the API and how to implement the model in an actual project, which are both necessary for acceptance and efficient use of the API. Thus, the use of this sample project in conjunction with the think out loud protocol allows measurement of the API's status with respect to the goals of reuse and rapid learning.

D. TEST PARTICIPANTS

There were three main qualities required of the test participants: they had to have experience in the implementation of Java interfaces; they had to have experience using Java APIs; and, they had to have basic understanding of Object Oriented Programming. To ensure these qualities were met, participants were screened using a questionnaire. The complete listing of the user group characteristics is presented in Table 5 located in Section 3 of Chapter IV.

A group of four subjects for the first test and five for the second were selected, with four of the second five returning from the first test. The subject group consisted of Java programmers with between one and five years of programming experience. All were Masters Students in a Computer Science Curriculum, with some also holding undergraduate degrees in Computer Science.

The wide range of programming experience was desired to provide various levels of feedback in response to the first time exposure to the API. The least experienced of the group struggled more with and made suggestions to improve learnability of the low level aspects of the API such as object interaction and method calls, whereas the most experienced programmers delved into actual design aspects of the API such as threads involved in dispatching and the multi-casting involved in the filter mechanism. This extreme range allowed for measurement of first time exposure for the entire spectrum of target users.

The quality of first time exposure is present in the test for two reasons. First, the API is new and it would require additional time to prepare subjects as experienced users of the Channel API; thus, the quality is practically unavoidable. However, it is also an important quality for measuring the success of the Channel API. It is the position of this paper that the impression made by the API during first time exposure has a great impact on reuse. If a programmer has a choice not to use the API and the API is hard to learn then he/she will likely seek alternatives. Thus, it is important that the participants in the test be first time users of the Channel API.

E. DATA ANALYSIS

Results appear in three forms, task completion, responses to end-of-test questions, and participant comments made during both task completion and post-task debriefing. The first two result forms are considered performance data and the third form is preference data. Each form is analyzed according to the procedures recommended by Rubin [Ref 3], which includes compilation, summarization, and pattern analysis of data. Each result form and the procedure used for its analysis are presented in the following subsections.

1. Task Completion

This performance data is in the form of binary test criterion responses. During task completion, the examiner observes a subject to determine if a series of test criteria are satisfied; the result of the observation, yes or no, is recorded on a data collection form. An example of this is whether or not a subject identified the correct method to use to complete a sub-task. Each of the criteria represents a necessary subject action for successful task completion.

Once all sessions for a specific test are complete, the results are grouped by sub-task and placed in a table with subjects as column headings and criteria as row headings. The binary results for each criterion are then analyzed according to the percentage of participants with positive responses and the percentage of participants with negative responses. These percentages give indication of task accuracy. Implications of necessary enhancements due to task accuracy are then discussed. This procedure is applied in Section B.1 of Chapter IV and V for Test 1 and 2 task completion data respectively.

The implications are discussed with respect to objectives for the specific task and the hypotheses made for its outcome. If the predicted outcomes are found true, then the enhancements proposed in the hypotheses are reconsidered, with possibly different changes arising. If the predicted outcome is found false, then the successful attribute is explained. For example, it was thought that the API's abstraction of event-driven programming would cause confusion, which did not occur.

2. End-of-Test Question Responses

This performance data is represented by subjects' question responses, some multiple-choice and some fill-in-the-blank. Responses from all subjects for all questions appear alongside the correct responses in Table 6 located in Section B.2 of Chapter IV.

This data is analyzed one question at a time. The purpose of each question is listed as usability attributes and metrics. Participant response statistics are then listed as the percentage of participants whose responses represent positive measurement of each purposed attribute. These statistics are followed by their implications for enhancements. This procedure is applied to data for Test 1 only and appears in Section B.2 of Chapter IV.

The implications are made with respect to memorability, learnability, comprehension, and perception of the Channel API. The questions are only used in the first test, so all of the listed attributes indicate first time exposure to the API. Specifically, they indicate problem areas that may deter novice and experienced first time users from continuing to use the API. The first group may discontinue use if they are not quickly able to comprehend the overall functionality of the API and the second group may discontinue its use if the API does not quickly explain itself.

3. Participant Comments

Participant comments are considered preference data. They may be very subjective in nature, yet when grouped, they give indications of problem areas, subject perception, and learnability issues. In addition, the comments may indicate why a particular sub-task suffers from a low task accuracy percentage. For example, a sub-task that required subjects to identify a sequence of method calls may suffer from a low accuracy percentage without an identifiable cause until review of a single comment indicates that the method descriptions conflict with the functionality implied by the method names.

To analyze the comments, they are placed into groups according to which aspect of the test they pertain to. The comments are used both as stand-alone input to design and documentation enhancements as well as together with the performance data to verify recommended enhancements. The stand-alone input arises from recurrent comments made by three or more subjects; this measure is adopted from the API analysis described in Chapter II Section B. The joint use of comments with performance data arises when apparently one time comments are found to support or explain results of task accuracy statistics.

F. DEVELOPMENT OF RECOMMENDATIONS

To determine required enhancements, the data analysis results are compared with test objectives and expected test outcomes. Each of the categories of test results outlined in previous chapters is considered jointly with respect to its impact on the test objectives. Implied enhancements resulting from task accuracy statistics, question response statistics, and participant comments are listed together in common categories. These categories are then screened for conflicts and considered with respect to impact on learnability and reuse.

The resulting enhancements are categorized as design or documentation enhancements. Design enhancements are those requiring changes to the API source code or in the actual structure of its underlying model. Documentation enhancements are those requiring changes to existing documentation and addition of new documentation. It is expected that documentation enhancements will out number the design changes. This is due to the lack of reference material for the API and hastily written source code comments. However, it is also expected that a portion of the API involving the *ChannelManagerAuthority* interface will require redesign.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. ANALYSIS OF CHANNEL API: BASIC COMPONENTS AND COMPONENT ASSEMBLY

This chapter presents the application of the analysis method outlined in Chapter III to the Channel API with emphasis given to testing its basic functionality.

A. TEST PLAN

This test plan was developed as described in Section B of Chapter III.

1. Test Purpose

The purpose of this test is to evaluate the Channel API to discover any portions of its basic functionality which may make the API hard to learn, use, or understand. The basic functionality is its underlying model, component assembly mechanism, and its use of event-driven data flow. Specifically, the API's class names, interface names, method names, and signatures are to be evaluated, where the signature is the return types and parameters of the method. In addition, the underlying Channel Model of the API is to be evaluated for learnability and perception. The test will also look to see if the API's abstraction of event based programming causes any confusion for those not familiar with event based programming.

2. Test Objectives

The following questions identify attributes of the Channel API which are to be measured. The answers to these questions will determine how usable the API is and how well it supports learnability and reuse.

- Are method names self descriptive? Can programmers ascertain their use by the names alone?
- Do method signatures convey proper use of methods?

- Do method signatures convey the method's purpose in the overall Channel Model?
- Do class names convey the object's purpose/role in the overall Channel Model?
- Are method names easy to remember?
- Are method purposes easy to remember?
- Are class purposes easy to remember?
- Is addition of a talker interface required to solidify the talker role in the Channel Model?
- For which aspects of methods and classes are documentation and use examples necessary?
- Is there a linear process to be followed when using the API to implement the Channel Model?
- Does API's abstraction of event-driven programming cause any confusion?
- Can users ignore event-driven programming?
- Does use of the term 'event' in method signatures cause confusion? Does it hinder learnability of the Channel Model? Does it cause errors in methods used?
- Does the *ChannelEvent* class cause any confusion?

3. Subject Profile

The following table describes the characteristics of participants in this test. The questionnaire used to screen is included in the appendices. To summarize, intermediate Java programmers were chosen in order to ensure that basic ideas such as classes and interfaces are already known as well as comprehension of how to use a Java API. The test subject profiles are contained in the table below.

	Subject 1	Subject 2	Subject 3	Subject 4
Programming Experience (Years: Y)	$2 < Y < 5$	$1 < Y < 2$	$5 < Y$	$2 < Y < 5$

Course in Object Oriented Programming (Yes/No)	Yes	Yes	Yes	Yes
Experience Using API (Yes/No)	Yes	Yes	Yes	Yes
Learning Style (Docs First, Try First, By Doing)	Docs First	Try First	By Doing	Try First
Documentation Style Preference (Online, Textual, None)	Textual	None	Online	None
Java Programming Experience (# of Projects Completed: P)	$5 < P$	$1 < P < 4$	$5 < P$	$5 < P$

Table 9. Test Subject Characteristics

4. Test Method

Four subjects were tested for this study. Each test took approximately fifty minutes. Subjects were given a questionnaire to determine if they met the user profile characteristics. Subjects who met the user profile were assigned a subject ID and all test forms were annotated with this. This facilitated protection of subjects' personal information (name and email address only).

API documentation was provided in the form of online HTML files generated by the Javadoc utility using the original source files as input. This documentation was installed by the test administrator prior to start of the test; these were bare minimum descriptions required for use; they only contained class/interface names and method signatures. The project write up (discussed below) included a brief introduction of the Channel API and the Channel Model.

A brief program project write up was given, followed by a series of tasks to accomplish the project. The subject was asked to describe the steps he/she would take to complete the tasks by referring to the Javadocs. Subjects' responses were video taped.

After completing the task series, some questions pertaining to the Channel API and documentation were asked to test memorability and comprehension of the API and Channel Model. Next, the subjects were polled for any subjective input to the use of the API using a series of scripted questions. Finally, subjects were debriefed, which included review of any problem areas noted during the session, review of comments made during the session, and review of the responses to subjective questions.

5. Task List

The following is a list of tasks for this test. For each task, there is a description of the goals, Test Criteria, and the hypotheses of the task outcome. The goals indicate what the task is measuring; the Test Criteria describe the indications for task completion; and, the hypotheses indicate the expected outcome to be proven or disproven.

During the actual test, the tasks are presented to the subjects in the form of a programming project write-up. Each task is further broken down into sub-tasks that direct the subjects toward investigating the specific functionality represented in the task. The project write-up is included in appendix A under the title *Project Write-Up*.

1. Design a class that is to be a channel talker.

Goals of Task:

- Determine if a talker interface is required.
- Determine if the subject realizes that definition of a data/message class is required and that any such class can be passed to the channel as the object event argument.
- Determine if subject understands the talker's interaction with the channel object.

Test Criteria:

- Subject has realized that there is no talker class or interface provided in the API and that any object can be a talker.

- Subject has discovered the talk methods in the *Channel* class and realizes that these methods are the talker's interface to the channel object.
- Subject has discovered the need for implementation of a data/message object.
- Subject presents a constructor for the talker class that requires a reference to a channel object.

<u>Hypotheses</u>:

- Subjects will be confused by the absence of a channel talker interface. This can be overcome with examples and sufficient documentation that explain and illustrate the ability of any class to fulfill the role of talker.
- Subjects will be confused as to how a talker object is able to talk on a channel. This can be overcome by giving an example in which a talker class contains a reference to a channel object passed into the talker class constructor.
- Addition of a *ChannelTalker* interface to the API will remove some of the confusion, but is not required.

2. Design a class that implements the *ChannelListener* interface.

<u>Goals of Task:</u>

- Determine if the purpose and use of the listener interface is understood by the subject.
- Determine if the *ChannelEvent* class causes any confusion.
- Determine if listener object's reference to the channel object is understood.

<u>Test Criteria:</u>

- Subject has found the *ChannelListener* interface.
- Subject has found the receive method in the listener interface.
- Subject is not confused by the *ChannelEvent* class argument in the receive method.

- Subject presents a listener constructor that requires a reference to a channel object.

Hypotheses:

- Subject will be confused by how a *ChannelListener* gains access to the channel object; an example showing a channel reference data member will fix this.
- The *ChannelListener*'s receive method argument '*ChannelEvent*' will confuse subjects. An example of the *ChannelEvent*'s purpose is required to show how it encapsulates the delivered object.

3. Design a class that uses a channel object to assemble the classes developed in task 1 and 2.

Goals of Task:

- Determine if the basic structure and functionality of the Channel Model is understood.
- Revisit the *Channel.talk* methods if they were missed during task 1.
- Determine if the flow of event objects from talker to channel to listener is understood.

Test Criteria:

- Subject finds/discusses the use of the the *Channel.addListener* and *Channel.addTalker* methods.
- Subject finds/discusses the *Channel.talk* methods.
- Subject finds/discusses the *ChannelListener receive* method.
- Subject discusses the *ChannelEvent* Class and its relation to the *ChannelListener*.

Hypotheses:

- Subjects will be confused by the talker-channel and channel-listener interactions. An example of the talker-channel and channel-listener interaction is required in the documentation.

- Subjects will be confused by the *ChannelEvent* object delivered to the *ChannelListener*. An example of the *ChannelListener* unpacking the *ChannelEvent* is required.

4. Complete a list of multiple choice and fill-in-the-blank questions.

   Goals of Task:

   - Test memorability of method and class names.
   - Test learnability of the Channel Model.
   - Test memorability of method signatures.
   - Test memorability of class roles.

6. Test Environment and Equipment

The test is delivered to subjects in a quiet library space. The API documentation is pre-installed on a laptop computer and is open before the test begins. The entire session, after preliminary paper work, is video recorded.

7. Test Monitor Role

The test monitor makes written recordings of notable comments that pertain to the specific Test Criteria for each task, as well as any unexpected useful comments. In addition, the test monitor acts as an online expert for questions that the subjects can not find answers to in the documentation. Once the examiner notices that all Test Criteria are met and/or a fair amount of time has passed for the task, the subject will be prompted to move on to the next task.

8. Evaluation Measures

The measurable test attributes are the completion of Test Criteria, End-of-Test Question Responses, and subject comments. These are evaluated using the techniques outlined in Section F of Chapter III.

The attributes of the API measured are comprehension, perception, readability, learnability, and memorability. These are measured using the subjects' responses and interactions with the API Javadocs. The Javadocs are currently the only representation of the interface; thus, one expected output is to note where further use examples and documentation are needed.

B. TEST RESULTS

The results from the tests were in the form of video tapes for each subject, the task list used by subjects containing possible responses, and the tables used by the examiner to record important Test Criteria and responses. Completion time for each of the four tests ranged from 45 minutes to 90 minutes. Review of each taped session side by side with the mentioned paper collections, required an approximate of 20 minutes in addition to the length of the tape. Compilation of the data resulting from the taped sessions and the data collection forms took approximately three hours, with its analysis requiring two additional hours.

Each session was reviewed in turn. The recorded responses were again checked against the Test Criteria for each task. Additionally, attention was given to any comments made by subjects to explain their confusion with, comprehension of, or perception of the currently tested API features. For example, any analogies used by the subjects to explain their perceptions of the underlying Channel Model were recorded as well as analogies used to explain the roles and functionality of the various model objects.

For each of the three tasks contained in the tests, subjects' responses were evaluated for:

- Input to Task Goals
- Meeting of Test Criteria
- Proof or Disproof of Task Hypotheses
- Unexpected Input

Additionally, the end-of-test questions were used to measure:

- Memorability of Class and Method Names
- Memorability of Method Signatures
- Memorability of Object Interactions
- Memorability of Object Assembly

The results of these two groups were used to determine implications for comprehension, learnability, and perception of the Channel Model, the Event Flow Model, and the API's Component Assembly Mechanism. Additionally, documentation points required to support rapid learning and comprehension of the API were determined. The following sub-sections contain discussions of the test results pertaining to task completion, end-of-test questions, and subjective comments in that order.

1. Summary of Task Completion

This section summarizes the data collected from subjects' completion each of the three tasks. The summary includes a listing of all sub-tasks, a table showing participants' meeting of Test Criteria, and implications of these results.

a. Task 1: Implementation of a Talker

This task prompts subjects to design a *SensorInput* class to perform the role of talker in the Channel Model. It is broken into four sub-tasks that focus subjects on specific test criteria.

1.1: Determine how the *SensorInput* class fulfills the role of Talker.

Test Criteria:

1. Subject comments on absence of a talker interface or class.

2. Subject realizes that any *java.lang.Object* can be a talker.

	Subject 1	Subject 2	Subject 3	Subject 4
Criteria 1	Yes	Yes	Yes	No

Criteria 2	No	Yes	Yes	No

Analysis:

The desire for this sub-task was to determine if subjects were confused by the absence of a talker interface and if they could determine that any generic Java Object could perform the role of talker. Three of the four subjects noticed immediately that the API did not provide any interface for the talker to implement and spent much time looking for an explanation in the documentation. The other subject looked through the documentation looking for a way to create a talker using a method within a given class.

Implications:

Since all the subjects either expected a talker interface to be provided or looked for an API mechanism to create a talker, quickly visible documentation must be provided to explain why no interface is provided and that contains an example of how any generic object can fulfill the role of talker. This will eliminate the initial confusion and will reduce the time required to produce a listener components.

<u>1.2: Determine if any of the API classes or interfaces need extension or implementation for this task.</u>

Test Criteria:

1. Subject expresses need or desire for a talker interface.

2. Subject determines that no API feature needs extension or implementation.

	Subject 1	Subject 2	Subject 3	Subject 4
Criteria 1	Yes	Yes	No	No
Criteria 2	Yes	Yes	Yes	Yes

Analysis:

The aim of this sub-task was to determine if there would be any benefits to providing a Talker interface and to determine if subjects could get past the absence by stating that no API feature was available. Half of the subjects wanted a talker interface to be provided, while the other half decided that it was not necessary. All of them were able to get past the confusion and conclude that no API mechanism was available for creating talkers.

Implications:

No interface is required. Providing documentation explaining how to define a talker will suffice.

<u>1.3: Determine how the *SensorInput* object will supply data to the *Channel.*</u>

Test Criteria:

1. Subject finds *talk* methods located in the *Channel* class.

2. Subject is confused by the *java.lang.Object* event argument in *talk* methods.

3. Subject comments on the need to define an event object.

	Subject 1	Subject 2	Subject 3	Subject 4
Criteria 1	Yes	Yes	No	Yes
Criteria 2	No	No	No	No
Criteria 3	Yes	No	No	Yes

Analysis:

There were two goals for this sub-task. The first was to determine if subjects could figure out how the user-defined talkers would interact with the API defined channels. The second was to determine if subjects could understand how

Talker's produced and pushed events to the channel and if the use of the word 'event' to describe the data pushed to the channel caused any confusion.

Three of the four subjects successfully found the talk methods in the *Channel* class for delivering events. The fourth was confused with how the events flowed through the model. None of the subjects were confused with the abstraction of events to deliver data to the Channel. Half talked about defining their own class to push through and the other half stopped at the decision to push *ChannelEvent* objects through.

Implications:

The mechanisms used by talker components to produce and push events to the channel are easily understood by examination of class and method descriptions; however, a short example will make it clear to all users.

1.4: What might a constructor for this class look like?

Test Criteria:

1. Subject describes a need for a *Channel* object reference.

2. Subject determines no constructor is necessary.

3. Subject's constructor contains a *Channel* object reference.

	Subject 1	Subject 2	Subject 3	Subject 4
Criteria 1	Yes	No	No	Yes
Criteria 2	No	No	No	No
Criteria 3	Yes	No	No	Yes

Analysis:

The goal of this sub-task was to determine if subjects could explain the precise relationship between instances of a talker and a channel that would be required for them to interact. The two possible arrangements are for both objects to exist within and be controlled by a larger class or, for the talker to contain a data member that references the channel it talks on. This is an advanced task that required subjects to give careful thought to how they would design the talker component and the main application it would be used in. The first arrangement was chosen by half of the subjects. The second was chosen by one with the last subject failing to immediately understand how the talkers and channels would become associated within an application.

Implications:

Although three of the four subjects were able to come to a design decision that would support the talker's association with the channel, the one failure was due to insufficient documentation of the talker-channel relationship. So, an explanation of the two possible arrangements and short examples are necessary.

b. Task 2: Implementation of a Listener

This task prompts subjects to design an *InputProcessor* class as a listener. Its purpose was to determine if subjects could find the API interface and methods necessary for a class to fulfill the role of listener, which includes receiving events from the channel and processing these events.

<u>2.1: Determine how the *InputProcessor* class fulfills the role of Listener.</u>

Test Criteria:

1. Subject finds *ChannelListener* interface.

2. Subject finds the *receiveEvent(ChannelEvent)* method.

3. Subject is confused by the *ChannelEvent* argument.

	Subject 1	Subject 2	Subject 3	Subject 4
Criteria 1	Yes	Yes	Yes	Yes
Criteria 2	Yes	Yes	Yes	Yes
Criteria 3	No	No	No	No

Analysis:

The purpose of this sub-task was to determine if subjects could determine the API mechanisms used for defining a listener component. This was accomplished by observing if subjects were able to find and understand the required interface. Additionally, it was desired to again see if the API's use of events caused any confusion. All of the subjects completed this sub-task successfully.

Implications:

The interface and method name are self-descriptive and allow programmers to quickly learn the listener role in the Channel Model.

2.2: Determine how an *InputProcessor* will interact with the *Channel.*

Test Criteria:

1. Subject identifies that the channel calls a listener's receive method.

	Subject 1	Subject 2	Subject 3	Subject 4
Criteria 1	Yes	Yes	No	No

Analysis:

The purpose for this sub-task was to determine if there was any confusion in the interaction between the channel and listener. Specifically, it was desired to see if the event-driven delivery mechanism caused any confusion.

Two of the subjects were not able to understand the interaction. They both knew that the events were supposed to be delivered to the listener by the channel, but were confused by how the events were delivered. From their comments, both were confused due to the *Channel* class' implementation of the *ChannelListener* interface. This implementation requires that the *Channel* define a *receiveEvent* method. After the examiner explained that this was an advanced feature that allowed a channel to be a listener on another channel, they were both able to overcome the confusion.

Implications:

An example and documentation is needed to explain the precise relationship of the two objects. Additionally, any hidden features such as the stacking of channels must be documented to eliminate confusing first time users. For this specific feature, an explanation in the *Channel* class comment should suffice.

<u>2.3: Determine how the *InputProcessor* is related to the *Channel.*</u>

<u>Describe a constructor for this class.</u>

Test Criteria:

1. Subject determines need for a *Channel* reference data member.

2. Subject provides a constructor with a *Channel* argument.

3. Subject determines that no constructor is necessary.

	Subject 1	Subject 2	Subject 3	Subject 4
Criteria 1	No	No	No	Yes
Criteria 2	No	No	No	Yes
Criteria 3	No	No	No	No

Analysis:

The purpose for this sub-task was to investigate the ability of subjects to comprehend the precise relationship required for channels and listeners to interact in the larger application. There are two possible arrangements. The first is for the main application to contain both objects and to mediate the association; and, the second is to place a *Channel* data member in the listener, which may be passed into its constructor. This was measured by the subjects' descriptions of the relationship necessary for the two objects to interact and their description of a listener constructor.

Three of the four subjects felt that the listener did not need direct access to the channel it listened on since it was the channel that initiated the event delivery. The fourth subject was again confused by the *Channel* class' implementation of the *ChannelListener* interface and became hung up trying to have the listener call the channel's *receiveEvent* method.

Implications:

The preferred method for providing the required relationship between the two objects is for the main program to provide the association since the channel initiates event delivery. An example demonstrating this will aide in efficient learning of the API. Additionally, it must be made immediately known that the listener is associated with a channel through a well known ID schema devised by the application designer. The importance of this ID is not well explained in the existing comments.

<u>2.4: How will a listener respond to data passed by the channel?</u>

Test Criteria:

1. Subject finds the *ChannelEvent* argument in the receive method.

2. Subject investigates the *ChannelEvent* class.

3. Subject finds the *getEvent()* method in *ChannelEvent* class.

4. Subject is confused by *ChannelEvent* argument.

	Subject 1	Subject 2	Subject 3	Subject 4
Criteria 1	Yes	Yes	Yes	Yes
Criteria 2	No	No	Yes	No
Criteria 3	No	No	Yes	No
Criteria 4	No	No	No	No

Analysis:

The purpose of this sub-task was to determine if subjects could understand the listener's responsibility to process the received events and that it was up to the programmer to define this processing. This also determined subjects' comprehension of the listener acting as the last stop in the event data flow model. These were measured by observing if the subjects found the *ChannelEvent* argument in the *ChannelaListener*'s *receiveEvent* method and if they were able to unpack this argument for processing.

Although none of the subjects were confused by the listener receiving a *ChannelEvent* argument, only one of the four discussed how to unpack this argument to extract the encapsulated event.

Implications:

Strong conclusions are not possible since the other three did not examine the *ChannelEvent* class functionality. However, since they all fit the same user profile with similar programming experience it conceivable that they would have understood how to extract the event encapsulated by the *ChannelEvent.* Thus, this requirement must be mentioned in the *ChannelEvent* and *ChannelListener* as well as in the API overview.

c. *Task 3: Component Assembly Using a Channel*

This task prompted subjects to design a *TrackingApp* class that uses a *Channel* to assemble the two classes they designed in the Task 1 and 2. The intent was to determine if subjects could find the API methods used to assemble components.

<u>3.1: Describe how you would create a *Channel* object.</u>

Test Criteria:

1. Subject finds *Channel* constructors.

	Subject 1	Subject 2	Subject 3	Subject 4
Criteria 1	No	Yes	Yes	Yes

Analysis:

This sub-task was designed to ensure that subjects understood that the channel is a stand-alone object that is already provided by the API. It is straight forward and only requires that subjects identify the constructor(s) used to instantiate a *Channel*.

By this point in the test, subjects had already looked through the *Channel* class documentation several times so it was predicted that this task would have 100% success; however, one of the subjects had found the *ChannelManager* class which led him to believe that he would not need to create *Channel* objects himself.

Implications:

The *Channel* is understood to be a stand-alone object which users may instantiate themselves. However, an example of its use and an explanation of the *ChannelManager* class would remove any confusion.

<u>3.2: Determine how to associate an *InputProcessor* with the *Channel.*</u>

Test Criteria:

1. Subject recalls that the *InputProcessor* is a *ChannelListenr.*

2. Subject finds the *Channel.addListener* methods.

	Subject 1	Subject 2	Subject 3	Subject 4
Criteria 1	Yes	Yes	Yes	Yes
Criteria 2	Yes	Yes	Yes	Yes

Analysis:

This sub-task was designed to ensure that the subjects understood the role of the listener class they defined and how to associate it with a *Channel.* It was successfully completed by all subjects.

<u>3.3: Determine how to associate an instance of the *SensorInput* class with the *Channel.*</u>

Test Criteria:

1. Subject recalls that the *SensorInput* is a talker.

2. Subject finds *Channel.addTalker* methods.

	Subject 1	Subject 2	Subject 3	Subject 4
Criteria 1	Yes	Yes	Yes	Yes
Criteria 2	Yes	Yes	Yes	Yes

Analysis:

This sub-task was designed for the same purpose as the previous one but with the implemented talker as the focus. It too was successfully completed by all subjects.

3.4: Determine in which class this association would be initiated.

Test Criteria:

1. Subject mentions a *Channel* data member in the talker.
2. Subject chooses main application where *Channel* resides.
3. Subject chooses talker class.

	Subject 1	Subject 2	Subject 3	Subject 4
Criteria 1	No	No	No	No
Criteria 2	Yes	Yes	Yes	Yes
Criteria 3	No	No	No	No

Analysis:

This sub-task was designed to evaluate the subjects' perceptions of where the Channel Model fits into the overall application design. This is determined by observing in which class they give control over registering their talkers and listeners. There were two possibilities. They could have the talkers and listeners use a data member which referenced a *Channel* or they could have the main application or class control the association.

By this point in the test session, they had all given more thought to the overall Channel Model and demonstrated this in their comments. Thus, they all chose to give control to the main class for adding talkers and listeners to the channel. This reaffirms the results from sub-task 2.3 in which all subjects placed control of interaction in the main class.

3.5: Determine how a *SensorInput* object will deliver data to a *Channel.*

Test Criteria:

1. Subject finds *Channel.talk* methods.
2. Subject is confused by *java.lang.Object* event arguments.
3. Subject is confused by the *talk* methods existing in the *Channel* class.

	Subject 1	Subject 2	Subject 3	Subject 4
Criteria 1	Yes	Yes	Yes	Yes
Criteria 2	No	No	No	No
Criteria 3	No	No	Yes	No

Analysis:

This sub-task was designed to determine if subjects experienced any confusion with the talker's event pushing functionality provided by the *Channel* class. In addition, it was to again look for confusion caused by the use of events. This was measured by observing subjects' descriptions of how to have the talker push an event to the *Channel.* The *Channel.talk* methods were found by all the subjects with no confusion caused by the event arguments.

3.6: Assume that a *SensorInput* has received a piece of input data.

Describe the sequence of method calls that occur to get the data from

the *SensorInput* to the *InputProcessor*.

Test Criteria:

1. *Channel.talk* method noted.

2. *InputProcessor.receiveEvent* method noted.

	Subject 1	Subject 2	Subject 3	Subject 4
Criteria 1	Yes	Yes	Yes	Yes
Criteria 2	Yes	Yes	Yes	Yes

Analysis:

This sub-task tests the event flow model used by the API. The methods used as Test Criteria are the essential calls needed for the various components to pass the events through the model. Each of the methods have already been seen and tested, but this sub-task puts them all together in relation to the assembled components. As expected, all subjects were able to successfully complete this task.

2. Summary of End-of-Test Questions

This section summarizes the data collected using the end-of-test questions, which participants were asked to complete without reference to the API documentation. Each question is listed with its correct response high-lighted. Statistics for participant responses and implications for the results are then discussed.

a. Listing of Questions

1. Which of the following methods is used for a Talker to send data to a Channel?

a. *ChannelTalker.talk(java.lang.Object talker, java.lang.Object event)*
b. *Channel.talk(java.lang.Object talker, java.lang.Object event)*
c. *ChannelListener.receive(java.lang.Object talker, java.lang.Object event)*
d. none of the above, programmer must write own method

2. Which interface or class does a Talker class implement or extend to participate in the Channel Model?

a. *ChannelTalker* interface
[✓ b]. java.lang.Object
c. *ChannelItem* class
[✓ d]. none

3. Provide the name of the interface that must be implemented by a class that is to receive data from a *Channel* object. What method must this class implement?

- *ChannelListener, receiveEvent(ChannelEvent event)*

4. When a listener receives data from a *Channel*, which object initiates the interaction?

[✓ a]. the *Channel* object
b. the Listener object
c. the *ChannelEvent* object
d. it is a system response to a Talker thread

5. What is the name of an object that is delivered to a Listener object by the *Channel*?

a. *ChannelData*
[✓ b]. *ChannelEvent*
c. *java.lang.Object*
d. Programmer must specify in Listener's receive method implementation

6. What class is the central object in the Channel Model?

- *Channel* class

7. What kind of object is delivered to the *Channel* by a Talker?

a. *ChannelEvent*
b. *ChannelData*
c. any *java.lang.Object*
d. none delivered
[✓ e]. Both a and c

8. How does a Talker or Listener object register with a *Channel* object? (Circle all that apply)

[✓ a]. *Channel.addTalker(java.lang.Object)*
b. *Channel.addTalker(ChannelTalker)*
c. *Channel.addListener(java.lang.Object)*

d. *Channel.addListener(ChannelListener)*

9. How does a Talker or Listener gain access to a *Channel* object for method calls?

a. Main program controls the access
b. A reference to the *Channel* in each class
c. Both a and b
d. No access possible
e. Do not remember

10. Can you briefly describe the structure of the Channel Model using words or a class diagram?

b. Participant Response Statistics

The table below shows the responses given by the four participants along side the correct responses. Below the table is a discussion of what each question was designed to measure and the implications participant responses have on the API design.

Q	Subject 1	Subject 2	Subject 3	Subject 4	Key
1	b	b	b	b	b
2	d	b	d	b	d
3	*ChannelListener*	*ChannelListener* *receiveEvent*	*ChannelListener* *receiveEvent* *(ChannelEvent)*	*ChannelListener* *receiveObject*	*ChannelListener* *receiveEvent* *(ChannelEvent)*
4	a	a	a	a	a
5	b	b	b	b	b
6	*Channel*	*Channel*	*Channel*	*Channel*	*Channel*
7	a	e	e	e	e
8	a, b, c, d	a, d	a, d	b, d	a, d

9	a	c	c	b	c

Table 10. Subject Responses to End-of-Test Questions

Question 1:

Purpose:

- Measure memorability of API method names by requiring recollection of the *Channel.talk* method.
- Measure learnability of the talker-channel interaction by requiring subjects to recall that reference to a channel is required to deliver it an event.

Participant Response Statistics

- 100% method name memorability
- 100% talker-channel interaction learnability

Implications

Based on the percentage of subjects who responded correctly, it is conclusive that the API's use of self-descriptive names is affective and that the talker-channel interaction is easy to learn.

Question 2:

Purpose:

- Measure memorability API fulfillment of the talker role by requiring subjects to recall that no API class or interface is provided for talker implementation.

Participant Response Statistics:

- 50% memorability

Implications:

Two of the four subjects correctly identified that no API feature was available. After reviewing the session tapes for the other two subjects, it was determined that their incorrect responses were due to the fact that a talker could be any generic Java Object, which was one of the response choices. Their choice of this other response indicates their comprehension of implementing the talker role; so, the question does not need to be thrown out and, it is concluded that the programmer's flexibility in implementing a talker component without the aide of an API class or interface is easily remembered.

<u>Question 3:</u>

Purpose:

- Measure memorability of API class names using the *ChannelListener* interface as a sampling.
- Measure memorability of API method names using the *receiveEvent* method as a sampling.
- Measure learnability of the *ChannelListener* role through recollection of its required receive-method.
- Measure memorability of API method signatures using the *receiveEvent* method signature as a sampling.

Participant Response Statistics

- 100% recall of interface name
- 50% recall of required method name
- 75% recall of listener role
- 25% recall of method signature

Implications

The results for memorability of names further confirm the success of the API designer's attempt to provide self-descriptive names that convey the purpose of the objects and methods. The high percentage of subjects that recalled the need for the

ChannelListener to provide a receive method confirms that the role of the listener in the Channel Model and the API feature used to fulfill the role are both easy to remember and easy to learn. The low percentage of recollection of the receive method signature is likely due to the short amount of time spent with the API. To aide rapid learning of method signatures, class diagrams and use examples are needed.

Question 4:

Purpose:

- Measure memorability of the channel-listener interaction by requiring recollection of the channel's initiation of the interaction.
- Measure learnability of the channel-listener portion of the event flow model used in the API.

Participant Response Statistics:

- 100% memorability
- 100% learnability

Implications:

These results indicate that the *Channel*'s initiation of event delivery to the listener is easy to remember and can be quickly learned by intermediate programmers in a short period of time.

Question 5:

Purpose:

- Measure memorability of API class names by requiring recollection of the *ChannelEvent* class.

Participant Response Statistics:

- 100% memorability

Implications:

This result indicates that the *ChannelEvent* class name, as a sample of API class names, is easy to remember and describes its purpose.

<u>Question 6:</u>

Purpose:

- Measure memorability of the *Channel* class' role as central object in the Channel Model.

Participant Response Statistics:

- 100% memorability

Implications:

The result of this question indicates that the Channel is understood to be the central object in the model. The question seems like a no-brainer; however, as will be discussed in the sub-section on subjective feedback, some of the subjects had difficulty with the *Channel* being a stand-alone object due to the existence of the *ChannelManager* class.

<u>Question 7:</u>

Purpose:

- Measure memorability of the two ways for a talker to push events to the *Channel* by requiring recollection of both delivery of a *ChannelEvent* and a generic Java Object.
- Measure learnability of the API contract provided for a talker to push events to a *Channel* through demonstrated recollection of one or both of the deliverable objects.

Participant Response Statistics:

- 75% memorability
- 100% learnability

Implications:

Three out of four subjects were able to recall both of the ways possible for an event to be pushed to a *Channel*. The one subject who did not choose both, chose a response that indicated the *ChannelEvent* class. These results indicate that the API feature provided for a talker to push events is easy to learn and remember in a short period of time.

<u>Question 8:</u>

Purpose:

- Measure memorability of the type of object associated with a channel as a talker through recollection of the proper *Channel.talk* method signature.
- Measure memorability of the type of object associated with the channel as a listener through recollection of the proper *Channel.talk* method signature.
- Measure memorability of the absence of a talker interface through non-choice of *ChannelTalker* as a method argument.

Participant Response Statistics:

- 50% memorability of talker type
- 75% memorability of listener type
- 50% memorability of talker interface absence

Implications:

One of the subjects chose all of the choices making his input unusable; however, he annotated the form to indicate that the *ChannelTalker* argument would be a Java Object by default. His annotation indicates that the first and third statistics may be 25% higher, but this is an assumption. Ignoring the unusable results still leaves strong implications that the subjects were able to remember the types of objects associated with the *Channel* and that no interface is provided. Documentation and an example of the three components and their interaction contracts will clear up any potential confusion.

<u>Question 9:</u>

Purpose:

- Determine the preferred method of controlling *Channel* access in an application through choice of where to place control.
- Measure perception of where *Channel* access is controlled in the overall application.

Participant Response Statistics:

- 25% put control in the main class/program
- 25% put control in the talker or listener class using a channel reference
- 50% felt either method was appropriate

- 25% perceived control of channel access in the main program
- 25% perceived control of channel access via a channel reference data member
- 50% perceived both ways possible

Implications:

Though the results for both of the measured areas are identical, they represent two different aspects of the API use. The first is an attempt to determine which of the two choices is preferred and the second is an attempt to determine if subjects realize that the two choices exist. In the first case, the results give no indication of a preferred method due to 50% of the subjects not making a solid choice. In the second case, the 50% indicates that the existence of two methods is not obvious. Thus, it is concluded that if either of the methods was intended over the other during API design, then this should be reflected in the documentation; and, if neither was intended over the other, then they should both be given equal coverage in the documentation.

Question 10:

Purpose:

Question 10 did not ask for a multiple choice response. Instead, it asked subjects to provide a description of the structure of the Channel Model using either words or a class diagram. The intent was to determine if subjects could, by the end of the 30 to 75 minute session, accurately describe the basic components of the model and their relationships.

Results and Implications:

Three of the four subjects were able to draw an accurate class diagram with the *Channel* class as an intermediary for the talker and listener components. The other subject indicated confusion with the *Channel* class' implementation of the *ChannelListener* interface and provided nothing more than this class-interface relationship in his diagram. His response may be due to the wording of the question. This is a reasonable assumption since his diagram was of the *Channel* and the interface he found it to implement. Overall, the strong 75% correct response rate indicates that the Channel Model structure is learnable in a short period of time.

3. Summary of Subjective Comments

Subjective comments were collected during task completion and during subject debriefing. In addition, comments were transcribed during review of the recorded sessions. Comments were categorized according to their relation to basic Channel Model components and component assembly, the event data flow model, programmer perception, or unexpected areas that were not the focus of this test session. The following sub-sections discuss these categories in the order listed.

a. Comments Pertinent to Channel Model and Component Assembly

All three of the tasks in the test were designed to illicit comments regarding the Channel Model components and their assembly. The first task investigated the role of the talker component in the Channel Model, the second task investigated the listener role, and the third task explored how they are assembled using a *Channel*. The focus in this area was to determine if first time users of the API could easily comprehend the basic components of the Channel Model, if the model was easily learned based on names and functionality, and if first time users could quickly comprehend how the API could be used to assemble components.

Overall, the subjects were able to comprehend the roles and responsibilities of the three basic Channel Model objects. However, their comments during to the task completion exposed the following areas which hindered comprehension and learnability of the model and its basic components.

- Missing Talker Interface

  Initially, two of the four subjects explicitly asked about a missing *ChannelTalker* Interface. The other two searched through the various class documentations looking for a way to create a *ChannelTalker* object. They commented that the documents provided a *Channel* class and a *ChannelListener* interface, but provided no indication of why there was no *ChannelTalker* interface available.

- Initial Model Overview

  During task completion, subjects asked for the documentation to immediately provide a general overview of the Channel Model to include all the objects, their roles, and how they interact. They described this overview in various ways, which probably reflects their various learning styles. One subject wanted a diagram or pictures showing the objects and their interactions. Another wanted one short example that displayed instantiation of each of the objects and the method calls made during their interaction. Another wanted simple explanations on a front page of each object with its role in the model and how to use it. One of the subjects expressed in the debriefing session that complete understanding was not possible without extensive use.

- Channel ID Assignment

  Three out of the four subjects did not understand where the Channel ID came from. They asked how the *Channel* created the ID and how the *Channel* would notify them of the ID assigned. It is clear from their comments that this important API functionality is not self-evident and must be well documented and noticeable upon first exposure to the API.

The ideas of Component Assembly and Interaction were generally understood by all of the subjects. However, they all fumbled with the specific details of using the API to assemble their components and to provide the desired interactions. Particularly, they presented varied descriptions of where in their programs they would create *Channel* objects and perform the registering of talkers and listeners. Subjects commented on the following areas during the test sessions.

- *Channel* Creation

  Lack of examples in the documentation caused confusion as to which class would create and control the *Channel* objects. Some of the subjects wanted to create channels in a main class that would keep track of them all but would not control the talker and listener interactions with the channels. The two basic ideas were to have the main class create the channels or to have a talker create the channel.

- <u>Component Assembly Point</u>

All of the subjects commented on confusion about where in the application the talker and listener components would be assembled. As in the channel creation above, they wanted to either have a main class create a channel and then add the talker or listener components to the channel or to pass a reference to the talker and listener components and have them add themselves.

- <u>Loose Association Between Talkers and Listeners</u>

At least one of the subjects expressed confusion about how talkers and listeners were associated. He stated that it was difficult to understand how a talker could push an event bound for a listener without knowing that the listener is ready to receive the event, will accept that type of event, or is even present.

b. Comments Pertinent to Event Flow Model

An important part of the API is its use of the Event-Driven data flow model. All three of the tasks exposed the subjects to this aspect and asked them to consider method calls and component relations involved. Below is a list that discusses subject comments that indicate confusion with Event-Driven data flow.

- <u>Agreement on Event Type</u>

Most of the subjects were comfortable with the talker component pushing any type of object to a channel and having it decide which Listeners to deliver the object to. However, one subject commented on the need for the talkers and listeners to agree on which type of event objects to push and receive. He viewed the two components as a single unit once associated with a channel and felt they needed to agree on the event type.

- <u>*Channel* Class Event Flow Method Names</u>

On one side of the Event Flow Model, is the lalker that pushes events to a *Channel* using a *talk* method. On the other side is a listener that receives events via a call to its *receiveEvent* method. One of the subjects expressed strong dislike for this pair of method names. He felt that the names did not match and wanted the *talk* method name to be changed to match the *receiveEvent* method name. He suggested the name be changed to *addEvent* on the talker side of the *Channel*.

- <u>*Channel* Class Add Method Names</u>

  One of the subjects liked that the API used the word event to define the object passed through from talker to channel to listener. He said that the word event helped him to understand how the data was passed through the model, especially its one-way aspect. However, he did not feel that the add methods contained in the *Channel* class clearly defined the roles of the talkers and listeners as event producers and receivers. He suggested changing the add method names to *addEventTalker* and *addEventListener.*

c. Comments Indicating Programmer Perception

The following comments and analogies describe the programmers' perceptions of the Channel Model, the API purpose, and the API usefulness. These are helpful in determining what analogies are useful in documentation and for determining what types of projects programmers will perceive the API is designed for.

- <u>*Channel* as a Stream</u>

  One of the subjects compared the *Channel* to a C-style Stream object with the talker dumping objects into it and the listener taking them off the other end.

- <u>Channel as a Buffer</u>

  One of the subjects described the channel as buffer with the talker filling it and the listener emptying it out.

- <u>Channel as a Socket</u>

  One of the subjects compared the channel to a Java Socket and compared the event flow model to the Client-Server model with the talker acting as the server and the listener acting as the client.

- <u>Channel as a Translator</u>

  One of the subjects thought that the name channel implied that it would provide some kind of translation service between objects passed between talkers and listeners. He speculated on how the channel might translate the objects into a format acceptable to the listener.

- Comparison to Pipes

One of the subjects described the channel as a pipe between the talker and listener. He also used the pipe analogy to overcome his confusion caused by the *Channel* class' implementation of the *ChannelListener* interface. He mentioned that it was similar to Unix-style pipes between processes.

- Duplex Communication

One of the subjects felt that the channel name implied duplex communication. However, he later expressed that the event abstraction led to a simplex or one way flow of data.

- Flexibility and Usefulness

Overall, subjects commented in their debriefing sessions that the API functionality would be useful and that it was a powerful tool.

- Intended Audience

All of the subjects, as intermediate level Java programmers, felt that with extensive use, the API functionality would become clear. One of the subjects did not feel that the API could be used to teach novice programmers. He stated that the documentation should reflect the intended audience and that for a general purpose API, documentation that taught basic programming ideas such as interfaces and Object Oriented Programming would hinder more advanced programmers.

d. Comments on Untested Features

Some of the comments made by the subjects did not pertain to any of the tested features but had notable implications for API use or design. Listed below is a summary of these comments and their implications.

- *Channel*'s Implementation of the *ChannelListener* Interface

Three of the four subjects were confused by the *Channel* class' implementation of the *ChannelListener* interface. For two of them, it caused confusion determining how the channel and listener interacted since both have a *receiveEvent* method. The examiner had to explain the purpose to eliminate the confusion. After this explanation, one of the subjects stated that this aspect confused the communication model.

- <u>Redundancy of *Channel* and *ChannelManager* Classes</u>

Two of the subjects made comments about the redundancy of the methods contained in the *ChannelManager* and *Channel* classes. They were confused about the purpose for the redundancy, which caused one of them to believe that the *Channel* could not be a stand-alone object and that a *ChannelManager* was required for all projects.

- <u>Class and Method Comments</u>

Many of the subjects expressed dislike for the class and method comments. Comments were made about the lack of detail in the comments. Some felt that the comments were more beneficial to the API designer than to an intended user. They wanted the comments to provide more detail on how the class and method is used.

C. RECOMMENDATIONS FOR API ENHANCEMENTS

The decision to make enhancements gave consideration to the task completion results, the end-of-test question results, and the subjective comments. These were analyzed with respect to API design and API documentation and their impact on learnability and comprehension of the API.

1. Design Enhancements

Only one design change was deemed necessary from the test results.

- <u>Change *talk* method name</u>

The lack of symmetry between the *talk* method and the *receiveEvent* method is of concern because it hinders both learnability of the event-driven data flow model and learnability of the roles of each class in the Channel Model. The receiveEvent name describes well the action of the listener in the event delivery model; but, the talk method, while describing the action of the talker, is part of the *Channel* class and should reflect its action as a queue instead. Thus, the *talk* method name should be changed to *addEvent*. This will also fix the misconception of duplex data flow implied by the *Channel* name. The add and receive methods imply opposite actions at each side of the channel, whereas the previous talk method did not necessarily provide implication of this one way flow.

2. Documentation Enhancements

- Channel Model Overview

A package overview will be added to the online documentation. This page will be the first page displayed when the documentation is activated. It will contain a simple diagram of the Channel Model components that indicate relationships and interactions; this diagram will embody component assembly and event-driven data flow. It will contain a listing of components with brief descriptions of their roles in the model and links to a short example that shows how all the components are instantiated and used.

- Application Component Contracts

A help page will be provided that explains how the API supports component contracts, which includes programmer control of Channel IDs, Filtering, and implementation of event objects.

- Explanation of Hidden Features

Features such as Channel implementation of the *ChannelListenr* interface must be explained to avoid incorrect speculation of their purpose.

- Class and Method Comments Improved

Provide explanation of Roles and purposes from user point of view to include useful analogies.

- Explanation of Important Channel Model Ideas

Channel IDs, Event-Driven data flow, Component Assembly not assumed to be known.

- Talker Page Link Added To Class Pane

A talker link will be added to the Javadoc class pane. This page will give an overview of the talker role. This will eliminate the confusion experienced by first time users.

V. ANALYSIS OF CHANNEL API: CHANNEL MANAGEMENT AND FILTERING

This chapter presents an analysis of the Channel API focusing on Channel Management and Filtering mechanisms.

A. TEST PLAN

This test plan follows the same development design as that used in the previous chapter.

1. Test Purpose

The purpose of this test is to evaluate two advanced features of the Channel API. First, comprehension, learnability, and perception of the roles and purposes of the *ChannelManager* class and *ChannelManagerAuthority* interface will be evaluated. In particular, points of confusion in the stand-alone use of both of these types of objects will be investigated as well as points of confusion that arise when the two objects are used together. Second, the event filtering functionality of the API will be evaluated.

2. Test Objectives

The following questions identify aspects of the API which are to be evaluated. Answers to these questions will determine where documentation is needed and where API design contributes to confusion. The objectives are broken into four categories: the first four relate to the *ChannelManager* class; the next four relate to the *ChannelManagerAuthority* interface; the following three relate to interaction of those two objects; and, the last three relate to the API's filtering process.

<u>*ChannelManager* Class Objectives</u>

- What documentation is needed for use of *ChannelManager* class?
- Can programmers understand when and how to use the *ChannelManager* class?

- What confusion arises when the *ChannelManager* class is used instead of directly manipulating Channel objects?
- What role or purpose is implied by the *ChannelManager* name?

ChannelManagerAuthority Interface Objectives

- What role or purpose is implied by the *ChannelManagerAuthority* interface name?
- Do the method names and signatures explain the interfaces' role and purpose?
- Can programmers understand when and how to implement the interface?
- Do programmers understand the responsibilities of the interface?

ChannelManager and *ChannelManagerAuthority* Objectives

- Can programmers discern the relationship between the *ChannelManager* class and the *ChannelManagerAuthority* interface?
- What confusion arises from this relationship? What documentation can fix this confusion?
- Do the name similarities cause confusion?

Filtering Objectives

- Do programmers understand the how and when to use the filtering interface?
- What confusion arises during implementation of the filter interface?
- Are the Channel Model's filterable attributes discernable?

3. Subject Profile

The subjects follow the same profile outlined in Chapter IV. Since advanced features are the focus of this test, the desired subjects are those from the first test. Reuse of the same subjects removes the initial confusion experienced during first exposure to the API. Furthermore, each of the subjects received a debriefing after the first test that summarized the correct use and basic functionality of the Channel API. In addition, one new subject was recruited. He was given an overview of the features covered by the first test session and the same briefing given to the subjects of that test. This fifth subject was added for two reasons. First, some of the results in the first test were inconclusive due to 50% positives and negatives. The fifth subject breaks this tie. Second, the new subject provides a first time exposure aspect to this second test that was present in the first.

Subject 5 had the following characteristics as outlined in Table 5 of Chapter IV Section 3:

- Between 2 and 5 years of programming experience
- At least one course in Object Oriented Programming
- Experience using an API
- Tries new concepts before searching for documentation
- No preference for support documentation style
- More than 5 projects using Java

4. Test Method

The same basic test method as Chapter IV was followed. However, it was expected that each of these test sessions would run longer at 90 minutes. The subjects were again given a task list to complete a number of tasks related to a sample programming project, which was a continuation of the previous write up.

To ensure that there was no confusion involving the basic functionality of the API, each session began with a brief overview of the Channel Model and its use. This involved an explanation of the API's use in fulfilling the first project write up. This explanation was scripted to ensure equality of information provided to each subject.

5. Task List

1. Use the *ChannelManager* Class for multiple channels.

<u>Goals of Task</u>:

- Determine if a single *ChannelManager* Object's replacement of multiple individual channel Objects is understood.
- Determine confusion that arises in this replacement.

- Determine subjects' perceptions of the role and purpose implied by the class name.
- Determine what documentation is needed to eliminate confusion and support quick comprehension of the Class' use.

Test Criteria:

- Subject demonstrates understanding of a *ChannelManager* Object's replacement of multiple channel Objects.
- Subject vocalizes confusion experienced due to the replacement.
- Subject verbalizes the role and purpose of the *ChannelManager* Class both implied by Class name and actual use determined by method names and signatures.
- Subject explains some documentation points that will clarify completion of this task.

Hypotheses

- Subjects will not understand the transition from channel Objects to a single *ChannelManager*. Sufficient documentation and an example will solve this.
- The redundancy of the *ChannelManager* Class methods and *Channel* Class methods will cause confusion.
- The *ChannelManager* Class name will imply the correct role and purpose of the class. This will aid in eliminating the initial confusion caused by the transition.

2. Implement the *ChannelManagerAuthority* Interface.

Goals of Task:

- Determine the confusion that arises from the interface name.
- Determine if the method names and signatures eliminate or enhance the confusion.
- Determine if the use and purpose of the interface is discernable.

- Determine if subjects can speculate on potential responsibilities of such an object for controlling channel Access.
- Determine documentation points to eliminate confusion.

Test Criteria:

- Subject describes the purpose and role implied by interface name.
- Subject investigates methods and comments on their implied purpose and use.
- Subject demonstrates increased or lessened confusion after method investigation.
- Subject verbally speculates the purpose and use of an interface implementation.
- Subject speculates on potential responsibilities of an interface implementation.
- Subject comments on useful documentation points.

Hypotheses

- Subjects will be confused by the interface name.
- The name will lead to false conclusion about the interface's role and purpose.
- Method names will not help to eliminate confusion.
- The interface purpose and use will not be correctly discernable.
- Implementation responsibilities will not be understood.
- A different name is needed to correctly reflect the intended use and purpose.
- A different name will eliminate some of the confusion.
- Examples and documentation will eliminate some confusion but will not completely overcome the name.

3. Use the *ChannelManager* (CM) Class with a *ChannelManagerAuthority* (CMA) Interface implementation.

Goals of Task:

- Determine if the CM's use of the CMA is understood.
- Determine what confusion arises from the relationship.
- Determine confusion caused by the name similarities.

Test Criteria:

- Subject discusses the appearance of the CMA in the CM constructor.
- Subject discusses confusion of the appearance.
- Subject discusses the CM's use of the CMA.
- Subject comments on the confusion caused by the name similarities.
- Subject discusses potential name changes.
- Subject discusses the relationship between the two objects.

Hypotheses

- The CM's use of a CMA will not be understood.
- The name similarities will cause confusion.
- The relationship between the two objects will not be correctly described.
- A name change will help with to alleviate the confusion.
- A contract for the interface is needed.
- A use example is required.

4. Implement the *ChannelFilter* Interface.

Goals of Task:

- Determine if the filtering process is understood.
- Determine confusion experienced when implementing the filter interface.
- Determine if the Channel Model's filterable attributes are apparent.

Test Criteria:

- Subject describes the use of an event filter.
- Subject discusses confusion that arises when using a filter.
- Subject discusses potential event attributes to filter by.

Hypotheses

- The filtering concept will be understood.
- Some confusion with how the filtering happens will arise due to the channel's control over the process. An example a filter implementation will eliminate this confusion.
- Filterable attributes will be understood, but an example will solidify the interface's flexibility.

5. Complete Multiple Choice questions.

Goals of Task:

- Measure learnability of *ChannelManager* Class.
- Measure memorability of *ChannelManager* Class purpose and use.
- Measure comprehension of when to use a *ChannelManager*.
- Measure comprehension of when to use a *ChannelManagerAuthority*.
- Measure comprehension of when to use both *ChannelManager* and *ChannelManagerAuthority*.
- Measure learnability of filtering process.

6. Evaluation Measures

As in the first test, comprehension and perception are the primary attributes to be measured. They will be apparent from subjects' reactions to class, interface, and method names.

B. TEST RESULTS

This section is a listing of the test results for Test 2; these include task completion statistics and subject comments.

1. Summary of Task Completion

This section summarizes the data collected from subjects' completion of the four tasks given in the project write-up. The summary includes a listing of all sub-tasks, a table showing participants' meeting of test criteria, and analysis and implications of their efforts with respect to the test criteria. Each task was broken into sub-tasks in the task list presented to the subjects. See the document entitled "Test 2 Task Write-Up" in the appendix to view the task list in the context of the Tracking Application project presented to the participants.

a. Task 1: Use of ChannelManager Class

This task prompts subjects to redesign the Tracking Application to utilize a *ChannelManager* for managing multiple *Channels*.

<u>1.1: Based on the *ChannelManager* class name, state the operations you would expect the class to perform.</u>

Test Criteria:

1. Subject speculates on adding and removing talkers and listeners to *Channels*.
2. Subject demonstrates confusion with respect to the class name and functionality.
3. Subject speculates on creating *Channnels*.

	Subject 1	Subject 2	Subject 3	Subject 4	Subject 5
Criteria 1	Yes	No	Yes	Yes	No
Criteria 2	No	No	No	No	No
Criteria 3	Yes	Yes	Yes	Yes	Yes

Analysis:

This purpose of this sub-task is to determine if the *ChannelManager* class name implies the functionality it is designed to provide. Specifically, it uses test criteria to check if the name implies that the *ChannelManager* creates *Channel* objects and associates talkers and listeners with them.

The test criteria results indicate that none of the subjects were confused by the intended basic functionality of the class. Three out of the five subjects were able to identify that the *ChannelManager* would both create *Channel* objects and associate talkers and listeners with the *Channel* using add methods; and, all five of the subjects decided that the *ChannelManager* would create *Channel* objects.

Implications:

These results indicate that the class name gives immediate indication of its intended functionality in the Channel Model.

<u>1.2: Based on the *ChannelManager* class methods, describe the purpose of a *ChannelManager*.</u>

Test Criteria:

1. Subject describes different role than task 1.1.

2. Subject comments on similarities and/or redundancy of *ChannelManager* and *Channel* classes.

3. Subject realizes that a single *ChannelManager* replaces multiple *Channels*.

4. Subject describes the *ChannelManager*'s role of controlling interaction between talkers and listeners using internal channels.

	Subject 1	Subject 2	Subject 3	Subject 4	Subject 5
Criteria 1	No	No	No	No	No
Criteria 2	No	Yes	Yes	No	Yes
Criteria 3	No	No	Yes	No	No
Criteria 4	No	No	Yes	Yes	No

Analysis:

The purpose of this sub-task is to determine if the method names, signatures, and comments allow subjects to quickly discern the role of the *ChannelManager*; and, if this role is different than the role implied by the name as discussed in 1.1. Understanding or of the role was measured with the meeting of the test criteria.

From Criteria 1, all of the subjects found the role provided by the class methods to be the same as that implied by the name. Criteria 2 and Criteria 3 are closely related. The first checks for subjects' recognition of the redundancy of *ChannelManager* and *Channel* class methods. The second checks to see if this redundancy leads them to understand that the *ChannelManager* replaces the need for multiple *Channel* instances. Three of the five subjects recognized the redundancy, but only one of these three understood the replacement. Criteria 4 checked for understanding that the *Channel* objects are not only managed by the *ChannelManager*, but exist internally to it. Only two out of five came to this conclusion.

Implications:

These results imply that the *ChannelManager* role is not quickly understood from examination of its methods. The existence of *Channel* objects internally to the *ChannelManager* and the manager's replacement of the need for multiple *Channel* objects must be understood right away to promote rapid learning. Thus, a short example and an overview document of the *ChannelManager* use with a short example is required.

<u>1.3: Determine how use of the *ChannelManager* class will affect the existing Tracking Application structure (refer to class diagram if needed).</u>

Test Criteria:

1. Subject determines that the *ChannelManager* will replace the *Channel* as the central object between talkers and listeners.

2. Subject realizes that direct access to the *Channels* is no longer required.

	Subject 1	Subject 2	Subject 3	Subject 4	Subject 5
Criteria 1	No	Yes	Yes	No	No
Criteria 2	No	Yes	Yes	No	No

Analysis:

The purpose of this sub-task is to determine if subjects understand how use of a *ChannelManager* will change the structure of their application. Specifically, that the *ChannelManager* will become the central object between talkers and listeners, and that they will no longer directly make calls to the *Channel* methods.

Only two of the five subjects were able to determine the correct program structure resulting from use of the *ChannelManager*.

Implications:

These results indicate that program design using the *ChannelManager* in place of multiple *Channel* objects is not understood. A design decision page must be introduced into the documentation that explains when and how to use a *ChannelManager* in an application. This page will quickly give designers an understanding of the distinctions in the two design options.

1.4: Determine how, where, and when *Channel* objects will now be created.

Test Criteria:

1. Subject determines that *ChannelManager* will create *Channel* objects internally.

2. Subject realizes that a *Channel* is created at the first call to *addTalker* or *addListener* methods with a specific channel ID.

3. Subject demonstrates confusion with this aspect.

	Subject 1	Subject 2	Subject 3	Subject 4	Subject 5
Criteria 1	No	No	Yes	No	Yes
Criteria 2	No	No	No	No	Yes
Criteria 3	Yes	Yes	Yes	Yes	Yes

Analysis:

The purpose of this sub-task is to force subjects to explain their understanding of the relationship between the *ChannelManager* and *Channel* class objects. Specifically, it checks to see if subjects understand the manager's encapsulation of the channels and if they understand which of the *ChannelManager* class methods causes creation of the channels.

All of the subjects demonstrated confusion with the creation of channels. Only two of the five understood that the channels were created internally to the manager; and, of these two only one realized that it happened when one of the add methods is called.

Implications:

The results indicate that the creation of channels by the *ChannelManager* is not understood from examination of the existing API documentation. The creation happens internally to the *ChannelManager*, so it is not an operation available to the users. However, subjects' knowledge of the existence of the *Channel* constructors leads them to expect to have control over channel creation. Thus, in the *ChannelManager* class comments it must be explained that users do not need to manually create *Channel* objects.

<u>1.5: Determine how the use of the *ChannelMangager* class will affect the structure of the *SensorInput* and *InputProcessor* classes.</u>

Test Criteria:

1. Subject realizes that each class will now contain a reference to the *ChannelManager* vice a *Channel*.

	Subject 1	Subject 2	Subject 3	Subject 4	Subject 5
Criteria 1	No	No	No	No	No

Analysis:

This sub-task asks the subjects to examine their talker and listener component implementations, which contain a reference to a *Channel*, and to determine if use of a *ChannelManager* will change the components' implementations.

It was a surprise that none of the subjects came to this conclusion. However, in later tasks they used calls to a *ChannelManager* object, which indicates that they understood the reference is necessary. Furthermore, these negative responses may be due to subjects' lack of understanding of the class diagram provided to assist in this task.

Implications:

No conclusions are possible from these results.

1.6: Describe how to assemble one pair of *SensorInput* and *InputProcessor* objects using the *ChannelManager*.

Test Criteria:

1. Subject creates a *ChannelManager*.
2. Subject manually creates the talker and listener components.
3. Subject invokes the *ChannelManager* add methods.

	Subject 1	Subject 2	Subject 3	Subject 4	Subject 5
Criteria 1	No	Yes	Yes	No	Yes
Criteria 2	Yes	Yes	Yes	Yes	Yes
Criteria 3	Yes	Yes	Yes	Yes	Yes

Analysis:

This sub-task requires subjects to demonstrate their understanding of the *ChannelManager* class functionality which they have gained with a brief investigation of the class. This checks to see if the class methods and comments are self-descriptive enough to allow comprehension and use of the class in a short period of time.

Three of the five verbally stated that they would need to create a *ChannelManager* object. The other two however, implied its creation with later calls to its methods. All of the five subjects created the talker and listener components and then assembled them using calls to the *ChannelManager* class add methods.

Implications:

These results indicate that an intermediate programmer can figure out how to use the *ChannelManager* in a short period of time based solely on the class methods. It does not however imply thorough understanding of the complete functionality of the class as is discussed in the previous sub-tasks.

b. Task 2: Implementation of a ChannelManagerAuthority

This task prompts subjects to redesign the Tracking application to utilize a *ChannelManagerAuthority* to aide in managing *Channels*. The intent is to determine what purpose is implied by the interface methods and name as well as to determine if subjects want to use the interface instead of the *ChannelManager.* Thus, two of the sub-tasks ask them to consider using the interface without the *ChannelManager*.

2.1: Based on the *ChannelManagerAuthority* interface name, speculate on the functionality you would expect an implementing class to perform.

Test Criteria:

1. Subject states that name does not give hint to functionality.

2. Subject discusses possible management of *Channels* similar to the role of *ChannelManager*.

	Subject 1	Subject 2	Subject 3	Subject 4	Subject 5
Criteria 1	No	No	No	No	Yes

Criteria 2	No	No	No	No	No

Analysis:

This sub-task is designed to check for initial confusion of the implied purpose of the interface and, to check if the name implies a role similar to that of the *CannelManager*.

Only one of the five subjects expressed confusion due to the interface name; and, none of them considered the interface name implied a management function.

Implications:

The interface name does not imply it will be used to manage *Channel* objects.

<u>2.2: Based on the interface methods, discuss the purpose for its use.</u>

Test Criteria:

1. Subject mentions enforcement of *Channel* access rules.
2. Subject mentions management of *Channels*.
3. Subject mentions same purpose as *ChannelManager*.

	Subject 1	Subject 2	Subject 3	Subject 4	Subject 5
Criteria 1	Yes	Yes	No	Yes	Yes
Criteria 2	Yes	No	No	No	No
Criteria 2	No	No	No	No	No

Analysis:

The purpose of this sub-task is to determine what purpose is implied by the interface method names, signatures, and comments. Specifically, it checks if the methods imply an access authority role or a management role.

Four out of the five subjects felt that the interface would be used to enforce access rules for *Channel* objects. Of these four, one of them considered this access control to be a management function. None of them considered this interface to be capable of the same role provided by the *ChannelManager.*

Implications:

Since the intended use of the interface was not access control [ref 1], the name clearly does not imply its intended use.

<u>2.3: Discuss the responsibilities an implementing class would have to fulfill.</u>

Test Criteria:

1. Subject mentions creation of *Channel* objects internally.
2. Subject mentions *Channel* management.
3. Subject mentions calls to *Channel* class methods.

	Subject 1	Subject 2	Subject 3	Subject 4	Subject 5
Criteria 1	No	No	No	No	No
Criteria 2	Yes	No	No	No	No
Criteria 3	Yes	No	No	No	Yes

Analysis:

This sub-task further investigates subjects' interpretation of the interface's purpose by asking them to explain how they would implement its methods. Specifically, it checks to see how they will reference *Channel* objects, if they decide to manage *Channel* objects within the authority implementation, and if they mention use of direct calls to *Channel* class methods by the authority.

None of the subjects thought that the authority implementation would create *Channel* objects. Only one subject felt the interface would manage channels; and, only two of the five felt that the authority would need to call *Channel* methods.

Implications:

These results indicate that there is no implied management of or association with the *Channel* objects by the authority implementation. This again implies that the purpose of the interface is not clear; and, a name change or precise documentation is needed.

2.4: Assume that you will not use a *ChannelManager* at all. Determine how you would use the *ChannelManagerAuthority* to manage *Channels*.

Test Criteria:

1. Subject discusses need to create and track *Channel* objects.
2. Subject determines need to create pass through methods that call *Channel* class methods.
3. Subject determines not to use the *ChannelManagerAuthority* for this puspose.
4. Subject determines that only access policy enforcement is possible.

	Subject 1	Subject 2	Subject 3	Subject 4	Subject 5
Criteria 1	No	Yes	Yes	-	Yes
Criteria 2	No	No	Yes	-	Yes
Criteria 3	Yes	No	No	Yes	No
Criteria 4	Yes	No	No	-	No

Analysis:

This sub-task asks subjects to forget about the possibility of using a *ChannelManager*, which was explored in Task 1, and to determine if the *ChanalManagerAuthority* would be beneficial to them for managing *Channel* objects in place of a *ChannelManager*.

Two of the five subjects, determined that they would not use the interface for management of channels. The other three subjects all determined that they could use the interface to track and manage channels. Of these three, only one felt that the authority would also mediate access to the *Channel* methods.

Implications:

These results imply that the subjects consider the interface to be useful for creating and tracking channels; but, it is not for mediating access to them as the *ChannelManager* does. This confirms that the manager and authority roles are considered distinct with non-overlapping functionality.

2.5: Describe how you would use the *ChannelManagerAuthority* to assemble one pair of *SensorInput* and *InputProcessor* objects.

Test Criteria:

1. Subject creates *Channel* object within the *ChannelManagerAuthority*.

2. Subject creates pass through methods to add talker and listener.

3. Subject determines this is not possible with the *ChannelManagerAuthority*.

4. Subject describes *ChannelManagerAuthority*'s use by an external class to enforce access rules.

	Subject 1	Subject 2	Subject 3	Subject 4	Subject 5
Criteria 1	-	No	No	-	-
Criteria 2	-	No	No	-	-
Criteria 3	Yes	No	No	Yes	Yes
Criteria 4	-	Yes	Yes	-	Yes

Analysis:

This sub-task builds on the previous by asking subjects to explore how the authority implementation would be used to assemble a talker and listener pair.

Implications:

Three of the five subjects correctly determined that assembly of components is not the intended purpose of the authority implementation. Additionally, three of the five incorrectly asserted that the purpose was to provide access control. This reconfirms the need to specify the intended use of the authority.

c. Task 3: Joint Use of the ChannelManager and ChannelManagerAuthority

This task prompts subjects to redesign the Tracking Application to use the *ChannelManager* class and the *ChannelManagerAuthority* interface together for managing multiple *Channel* objects. This explores the implied use of the authority by the manager as is indicated by the *ChannelManagerAuthority* argument in on of the *ChannelManager* constructors.

3.1: Discuss how the *ChannelManager* uses the *ChannelManagerAuthority*.

Test Criteria:

1. Subject uses the *ChannelManagerAuthority* to enforce access rules.

	Subject 1	Subject 2	Subject 3	Subject 4	Subject 5
Criteria 1	Yes	No	Yes	No	Yes

Analysis:

This sub-task takes a poll of the number of subjects who conclude that the purpose of the authority interface is to provide access control, which three out of five subjects did.

Implications:

It was known during the design of this test, that the interface name and its method names implied access control but its comments implied some kind of management function. After exposure to the possibility of managing channels in Task 2, this sub-task confirms the expected conclusion with respect to the purpose of the authority interface; and, it confirms the need to document the intended use of the *ChannelManagerAuthority* interface.

3.2: Discuss the similarities between the two names and the impact it has on implied uses.

Test Criteria:

1. Subject is confused by the similarity of the names.

2. Subject expresses a need to change the *ChannelManagerAuthority* name.

	Subject 1	Subject 2	Subject 3	Subject 4	Subject 5
Criteria 1	No	No	No	No	No
Criteria 2	No	No	No	Yes	Yes

Analysis:

This sub-task asks subjects to consider the names of the *ChannelManager* class and *ChannelManagerAuthority* interface along with their conclusions about their uses and to express any confusion caused by the similar names. Specifically, a name change for the authority based on their brief exposure is desired.

Unfortunately, only two of the subjects desired a name change even though their responses in previous tasks indicated incorrect conclusions to the intended use.

Implications:

The name similarities do not cause confusion and is not the cause of the incorrect conclusions of intended use.

3.3: Discuss if use of the two objects together changes the way you would implement the *ChannelManagerAuthority*.

Test Criteria:

1. Subject indicates that role of interface is changed by joint use.

	Subject 1	Subject 2	Subject 3	Subject 4	Subject 5
Criteria 1	No	Yes	No	No	No

Analysis:

This sub-task asks subjects to compare their use of the authority interface in Task 2 and its joint use with the *ChannelManager* in this task and to determine if a different role is implied.

None of the subjects changed their implementations. The one who indicates yes in the chart thought the question was with respect to the design of the interface itself.

Implications:

The interface is seen to have the same purpose whether it is used with a *ChanneManager* or by itself.

d. Task 4: Implementation of a ChannelFilter

This task prompts subjects to consider the use of a filter that will only allow a *Channel* to deliver desired events to a single *InputProcessor*. The intent is to have the subjects investigate the use of the *ChannelFilter* interface and ensure that its documentation, functionality, and intended use in the Channel Model are understood.

<u>4.1: Determine which API feature you would use to create a filter.</u>

Test Criteria:

1. Subject determines to use the *ChannelFilter* interface.
2. Subject discusses the pre-defined filter provided in the API.

	Subject 1	Subject 2	Subject 3	Subject 4	Subject 5
Criteria 1	Yes	Yes	Yes	Yes	Yes
Criteria 2	No	No	Yes	Yes	Yes

Analysis:

This sub-task determines if the API's available filter mechanism, namely the provided interface, is immediately apparent. It also checks to see if subjects investigate the predefined filter implementation as an example of the filtering process.

All of the subjects found the provided interface and three of them examined the example implementation.

Implications:

This task was mainly to prompt subjects to examine the filtering mechanism. It was expected that the *ChannelFilter* interface name would be self-descriptive. The results indicated that the designer's focus on providing self-descriptive names for the filtering was effective.

4.2: Give an overview of the filtering process. Include the roles of the *SensorInput, Channel, InputProcessor,* and filter objects.

Test Criteria:

1. Subject realizes that the *SensorInput* only produces events.
2. Subject realizes that the *Channel* uses the filter to screen events for delivery to listeners.
3. Subject comments on the filters job of defining the filter process.
4. Subject realizes that the *InputProcessor* only receives events.
5. Subject tries to have the filter deliver events.
6. Subject expresses confusion with this task.

	Subject 1	Subject 2	Subject 3	Subject 4	Subject 5
Criteria 1	Yes	Yes	Yes	Yes	Yes

Criteria 2	Yes	Yes	Yes	Yes	Yes
Criteria 3	Yes	Yes	Yes	Yes	Yes
Criteria 4	Yes	Yes	Yes	Yes	Yes
Criteria 5	No	No	No	No	No
Criteria 6	No	No	Yes	No	No

Analysis:

This sub-task determines if the filtering mechanism is self-descriptive. Specifically, it ensures that subjects can ascertain the filtering process quickly based on the provided interface and the example implementation.

All of the subjects were able to quickly understand how the Channel Model objects interacted when filtering is employed. Only one subject expressed confusion with this task. He was unclear about when the filtering occurred. This will be explored more in the next section dealing with subjective comments.

Implications:

The place of filtering in the Channel Model is easily understood. The design of the filtering mechanism is efficient.

<u>4.3: Discuss possible attributes of the four objects mentioned in task 4.2 which you could use to do the filtering.</u>

Test Criteria:

1. Subject mentions class names.
2. Subject mentions priorities.

3. Subject mentions time stamp on event.
4. Subject mentions user defined attributes.
5. Subject mentions attributes for all objects.

	Subject 1	Subject 2	Subject 3	Subject 4	Subject 5
Criteria 1	No	Yes	Yes	Yes	Yes
Criteria 2	No	Yes	Yes	No	No
Criteria 3	No	No	No	No	No
Criteria 4	No	No	Yes	No	Yes
Criteria 5	No	Yes	Yes	No	Yes

Analysis:

This sub-task checks to see that subjects understand how the filtering is accomplished. Specifically, it checks verifies that they can discern the filterable attributes of the various objects.

The specific test criteria indicate the level of investigation subjects gave to this task. Whether they found all the filterable attributes is not critical. Criteria 1 and 2 results indicate that most of the subjects could discern the available attributes such as class names and priorities; and, Criteria 5 results indicate that most subjects discerned that all objects possessed filterable attributes. However, only two of the five mentioned the flexibility of user defined attributes which are not related to the API implementation of the objects. For example, a data member created for filtering in a talker, listener, or event implementation.

Implications:

Subjects understand the API defined filterable attributes; but, the flexibility of user-defined attributes is not apparent.

4.4: Discuss the method calls that occur when a C*hannel* uses the filter.

Test Criteria:

1. Subject finds the *isAccepted* method in the *ChannelFilter* interface.

2. Subject mentions internal methods of the implemented filter.

	Subject 1	Subject 2	Subject 3	Subject 4	Subject 5
Criteria 1	Yes	Yes	Yes	Yes	Yes
Criteria 2	No	No	Yes	Yes	No

Analysis:

This sub-task was designed to check that subjects could describe the filter process with specific method calls, thereby demonstrating understanding of how all the objects interact in the process. Specifically, realization that the *Channel* object calls the filter's *isAccepted* method indicates understanding that the channel relies on the filter to provide the filtering capabilities.

All of the subjects correctly demonstrated the required method calls. However, only two went further to discuss the internal calls made by the filter.

Implications:

The *Channel* objects dependence on the filter is understood.

2. Summary of Subjective Comments.

The test considered two different API features. The first tested class and interface provided for channel management and the second tested the filtering mechanism. The comments are grouped with respect to these two categories.

a. Channel Management

The following comments highlight participants' confusion with the use of the *ChannelManager* class, the use of the *ChannelManagerAuthority* interface, and their joint use.

- *ChannelManagerAuthority* Control Over *ChannelManagers*

One of the subjects felt that the word *Authority* in the name implied that this interface would provide another layer of control hierarchy, with the authority having control over many instances of *ChannelManager*.

- Inconsistent Terminology

The *ChannelManagerAuthority* class comments describe a channel controller component; this term is not used anywhere else in the documents. In addition, the *ChannelManager.getAuthority* method states that it provides access to the 'Security Authority'. One of the subjects felt that all these differences confused the purpose of the authority interface.

- Channel Creation

Many of the subjects wanted the *ChannelManager* class to have a method for creating *Channel* objects. They felt that the class methods implied existence of channels but nothing in the documentation explained how they were created. They felt it was essential to know when they were created before a user could begin to make any of the other method calls such as *addTalker* or *talk*. They spent much time searching the *Channel* class documentation looking for a *CreateChannel* method. This led one subject to incorrectly believe that the channels existed externally to the manager. They expressed the desire to have control over *Channel* creation or at least an explanation of how it happened.

- *ChannelManager/Channel* Relation

A few of the subjects were confused by the relationship between these two objects. They thought that channels existed externally to the manager and that the manager only kept an internal listing of channels. One subject related this to the word 'Manager' in the name, which he felt did not imply a creation role. The subjects spent much time trying to figure out how to associate their *Channel* objects with the manager.

- Get-method Without Matching Set-method

One subject was looking for how the *ChannelManager* stored *Channel* objects and found the *getMemberChannels* method. He immediately asked where the *addChannel* method was; this is an example of a violation of programming discourse rules [ref 2]. He explained that is was the get was missing a matching set.

- *ChannelManagerAuthority* Comments

All of the subjects felt that the method descriptions in the interface did not match with the function implied by the method names. One example was a get method with a comment that described a set operation.

- *ChannelManagerAuthority* Purpose

All of the subjects stated that they could not figure out what the designers intended use was for this interface. Most of them thought that it was for enforcing access control rules. One thought that it had both access control and scheduling responsibilities. One of them felt that it was an un-necessary interface that checks performed by its methods were already done by the application programmer when channels were created and talkers and listeners were associated. All of them concluded that the purpose needed to be clarified.

- *ChannelManagerAuthority* Method Arguments

One of the subjects felt that the interface method arguments were misleading with some indicating the interface is tied to one channel and others indicating authority over multiple channels. He explained that the *getListenerPriority* method took a class name argument; he wanted to know how the authority would return a priority for a specific listener if multiple listeners of the same class were registered with a channel. Furthermore, he asked what would happen if a class of listeners were registered with many channels; how would the authority know for which channel to return the priority for? The interface also conflicts with some of the API's *addListener* methods that allow assignment of a priority to a single listener. Whereas the authority interface's *getListenerPriority* method gets a priority for an entire class of listeners. He felt these problems indicated poor design of the interface.

- *ChannelManagerAuthority* is Redundant

One of the subjects felt that the checks performed by the interface were redundant. He explained that it made no sense to check things that you had already checked during design. He gave the example of creating a channel, authorizing talkers and listeners, and then checking your own authorization. To him it made no sense.

- *ChannelManagerAuthority* Name Change

One of the subjects commented that the word 'Manager' should be taken out of the name because that it implied authority was over the *ChannelManager* and not an individual *Channel*. He suggested it be changed to *ChannelAuthority*. Another subject, who thought that the interface is used for access control, wanted to change the name to *ChannelSecurity*.

b. Filtering Process

- Bad Approach to Filtering

One subject felt that the use of filtering in the *Channel* was a bad approach. He felt that events should just be pushed to all listeners. He felt that filtering still required examination of all listeners to check their filters and that this eliminated any benefit gained from not delivering to a particular listener.

- Filter Association

One of the subjects spoke of the filter as a single object employed by the channel on its receiving side that would drop incoming events that the channel did not want to accept. Two other subjects did not immediately understand that the filter was associated with a specific listener and that their may be multiple filters for a listener. Only the filter manipulation methods removed their confusion.

- *ChannelFilter* Interface Comment Confusing

One of the subjects felt that the interface comment was poorly written and appeared to contradict itself at first reading.

- Filter Eliminates Broadcast

One of the subjects did not like the fact that after choosing to use a filter would eliminate the reception of all other events. He thought this would require definition of a filter for every type of event a listener required.

- Inconsistent Method Names

One subject noticed that the *Channel* class contained an *addFilter* method whereas the *ChannelManager* class contained an *addListenerFilter*. He felt that if the functionality is the same than they should have the same names.

- Filter Application Order

One subject wanted to know in what order the filters would be applied. He could not find an answer in the documentation and thought it should be explained.

C. API ENHANCEMENTS

The enhancements are categorized as design changes or documentation changes. They are discussed according to their impact on comprehension, learnability, and perception of the API and its intended use. The majority of the enhancements are related to documentation of features and their use. However, there are some enhancements for the design of the *ChannelManagerAuthority* interface.

a. Design Changes

- Addition of *ChannelManager.createChannel* Method

The absence of a *createChannel* method violates a programming discourse which leads to confusion at initial exposure to the class. Programmers expect set and get methods to occur in pairs. The apparently missing *create* method impacts class learnability. Addition of a create method will not only eliminate the confusion, it will also enhance comprehension of the Channel Model. The create method will give programmers control over channel creation and will make the necessity of a *Channel* object between talkers and listeners more apparent.

- *ChannelManagerAuthority* Name Change

The name of this interface must be changed to reflect its purpose in the API. The current name implies some kind of management or access control function;

however, the intended use was to provide a programming tool to ensure that components are assembled correctly. The new name must indicate this clearly.

- *ChannelManagerAuthority* Method Arguments

The level of checking implied by the types contained in the method arguments does not match with the attributes associated with objects during component association. For example, when a listener becomes associated with a *Channel*, it includes a priority; however, the *ChannelManagerAuthority* screens based on the priority of an entire class. The intended use of the class must be thoroughly specified and the methods redesigned to ensure the functionality is available.

- Possible Elimination of *ChannelManagerAuthority*

After consideration of the above two design changes, it may be concluded that this interface is not necessary. The intention of the interface must be clearly stated; then, it must undergo further usability analysis to determine if it is a necessary and useful feature of the API.

b. Documentation Changes

- *ChannelManager* Use Description

Both the *ChannelManager* and the *Channel* class comments must include a discussion of the design changes that occur when a *ChannelManager* is used. It must stress the differences between a project using multiple channels and a project that allows the *ChannelManager* to manage the multiple channels. This will enhance learnability design options.

- *ChannelManagerAuthority* Method Descriptions

The method descriptions found in this interface must be changed to describe the same functionality as that implied by the names. The disparity between the two implied functionalities greatly impacts comprehension and learnability of the interface.

- Class Comments

All class comments, including method descriptions, must be rewritten to describe the roles of the classes in the Channel Model and the purpose of their methods in fulfilling those roles. This will greatly enhance quick learning and comprehension of the API.

- Description of *ChannelManager/ChannelManagerAuthority* Interaction

The interaction between these two objects must be thoroughly explained using both visual diagrams and written documentation. This is necessary to ensure comprehension of how the manager uses the authority and the different behavior expected when an authority is used.

- *ChannelManagerAuthority* Purpose

The purpose for implementing this interface must be clearly stated in the API overview documentation and in the interface comments.

- Filtering Overview

The filtering mechanism is easy to learn; however, the object relationships that occur when filtering is used are not apparent. An overview of the filtering implementation and process will enhance learnability of this feature. This overview must include the association between the filter and listener object, the use of the filter by the channel, and the order of application when multiple filters exist for a single listener.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSIONS AND FUTURE WORK

Overall, the Channel API was well received by the test subjects. All of them were able to arrive at a basic understanding of the API's use and underlying concepts during test sessions ranging from 40 minutes to 90 minutes. They made comments suggesting that they were satisfied with the intent of the API and that they would use the API for programming projects. However as was discussed in the Results Section of the two previous chapters, the tests revealed some minor design flaws in the API that needed to be addressed. Implementation of the recommendations is discussed in the following section. In addition, the necessity for future work to follow up on the changes to the API as well as to test the API in its intended component based design setting are discussed in Section B of this chapter.

A. CHANNEL API

The changes to the API discussed here combine the recommendations for the test results presented in Chapters IV and V. Additionally, the method of API distribution is briefly mentioned.

1. Changes Made

Changes to the API involved additions or improvements to the documentation and additions or improvements to the source code.

The necessary document changes as recommended in the test chapters are satisfied by the following:

- A package overview page is provided that lists the API classes/interfaces with their roles in the Channel Model and links to use examples, provides a diagram of the Channel Model objects and their interactions, and explains important API concepts such as the Channel ID schema, filtering, and event flow.
- Rewriting of the source code comments is accomplished to ensure that the Javadocs they produce will contain accurate and useful descriptions of the classes, interfaces, and methods.

The design changes made are as follows:

- *Channel.talk* method is renamed to *Channel.addEvent* to reinforce the simplex event flow and to provide a balance between the actions occurring on either side of the channel.
- *ChannelManager* class now contains a *createChannel* method to eliminate confusion caused by apparent programming discourse rule violation.
- *ChannelManagerAuthority* interface name is changed to *ChannelAccessControl* to better describe the intended functionality of the interface.

2. API Distribution

The Channel API is located on the Web at www.SAAMNET.org/ChannelAPI.html. The class files, documentation files, and source code files are available for download either as an entire package or as separate pieces. It is recommended that the documentation be downloaded for reference during use of the API since the background section of this thesis does not cover all the methods and classes contained in the package.

B. FUTURE WORK

1. Feedback from Large-Scale Use

The Channel API is designed to support Component Based Design (CBD) projects. The testing done in this paper only looked at general usability attributes and did not consider the specific qualities of CBD support. Thus, feedback from the API's use for these types of projects must be obtained. Distribution of the API should be tracked and feedback obtained from the users to find any quirks experienced during its use in CBD projects. This feedback should include well planned questionnaires that ask for comments on specific attributes of the API.

Additionally, further usability testing should be done on a larger scale focusing entirely on CBD support. This would require changes to the testing method used in this paper. Three changes to the approach are suggested. First, the project should be an actual real world project that uses CBD concepts in a large-scale project, which would include a large number of components and cooperation between a large-number of programmers for their assembly. Second, the testing should not consist of single one-time sessions per subject; instead, testing should be done in-line with the project implementation, which will allow observance of the API's use from initial exposure through project completion. Third, consideration should be given to using the think out loud protocol with paired programming; this will make the subject verbalizations more natural and will eliminate intrusions made by the test monitor to extract feedback. These changes will allow more rigorous testing of the API.

2. Redesign and Retesting of *ChannelAccessAuthority*

The changes made to the *ChannelAccessAuthority* interface, previously the *ChannelManagerAuthority*, were minimal. For example, the interface comments and its methods' comments were changed to provide a better indication of its use. However, there are still questions with respect to the parameters required in the interface methods. Some of these parameters do not seem to provide the level of access to components suggested by the method name. For example, the parameter in the method *getListenerPriority* provides access to a listener based on a String class name, which implies that the priority is associated with an entire class; but, the priority is associated with a single listener when it registers with a channel.

These questions are due to design problems of the interface. Specifically, its purpose in the overall Channel Model is not well planned. To overcome this problem, the attributes of the components this interface operates on must be examined to determine better parameters for the interface methods. Once this is accomplished, further usability testing can be done focusing entirely on the redesigned interface.

3. Generalization of the Channel Model as a Pattern for Component Based Design

The discussion of the Channel Model provided in Chapter II Section B is a limited example of how the Channel Model may be generalized allowing discussion of the model with out respect to any particular programming language. Future work is possible to determine if the model is capable of providing all the needs of a component based design pattern for assembling components in a single application. If all the needs are not met, then the deficiencies could be noted and the model adjusted to add the missing requirements. The API could then be redesigned to show proof of concept for the resulting pattern.

In addition, extension of the Channel Model for assembly of components across process boundaries has potential for future work. Specifically, determination of the requirements for the Channel Model to allow inter-process message passing and the work necessary to fulfill these requirements must be determined. If there is any benefit in extending the API in this manner, then the work may be accomplished.

4. Investigate Making *Channel* class Private

There was some confusion experienced by all of the test participants due to the redundancies of the *Channel* and *ChannelManager* class methods. The confusion is currently mitigated by the introduction of explanatory documentation. However, it is thought that the *Channel* class could be made a private class. This would make the *ChannelManager* class the central point for channel access in an application. This change must be considered to determine the resulting changes required to the *ChannelManager* class and to determine if elimination of access to the *Channel* class by programmers would have a negative impact on the extensibility of the Channel API.

APPENDIX A

TEST 1 PROJECT WRITE-UP

This section contains the project write-up presented to participants of test 1.

Contact Tracking Application Project

You are assigned to develop a shipboard contact tracking application that receives inputs from sensors and processes these inputs. There are two main classes to be developed. The first is a SensorInput class that sends the input to the appropriate processor. The second is an InputProcessor that receives and processes the inputs.

You are to develop these classes independently but they must be assembled in some way that facilitates the passing of information. To do this you will use the Channel Model API. This API provides a Channel object that will tie together the two classes described above. The Channel model works by defining objects as either Listeners or Talkers, each of which is added to the same Channel. A talker passes data to the Channel and the Channel delivers the data to a listener.

For this session, you will examine the API documentation to determine how you would accomplish the project. As you follow the task list presented below, please verbalize your thought process while examining the API documentation. Describe the classes, interfaces, methods, and method arguments you would use and, describe any confusion you encounter with the API or its documentation. Also, comment on any functionality for which examples or further explanation would be useful.

Task1: Design the SensorInput class as a Talker.

1.1 Determine how the SensorInput class fulfills the role of Talker.

1.2 Determine if any of the API classes or interfaces need extension or implementation for this task.

1.3 Determine how the SensorInput object will supply data to the Channel.

1.4 What might a constructor for this class look like?

Task2: Design the InputProcessor class as a Listener.

2.1 Determine how the InputProcessor class fulfills the role of Listener.

2.2 Determine how an InputProcessor will interact with the Channel.

2.3 Determine how the InputProcessor is related to the Channel. Is either object a data member of the other class.

2.4 How might this class constructor look?

2.5 How will an InputProcessor handle the data passed by the Channel? Explain the processing of the object passed to the listener.

Task3: Design the TrackingApp class that uses a Channel to assemble the two classes you designed above.

3.1 Describe how you would create a Channel object.

3.2 Determine how to associate an InputProcessor object with the Channel object.

3.3 Determine how to associate an instance of the SensorInput class with the Channel object.

3.4 Determine in which class this association would be initiated.

3.5 Determine how a SensorInput object will deliver a data object to the Channel.

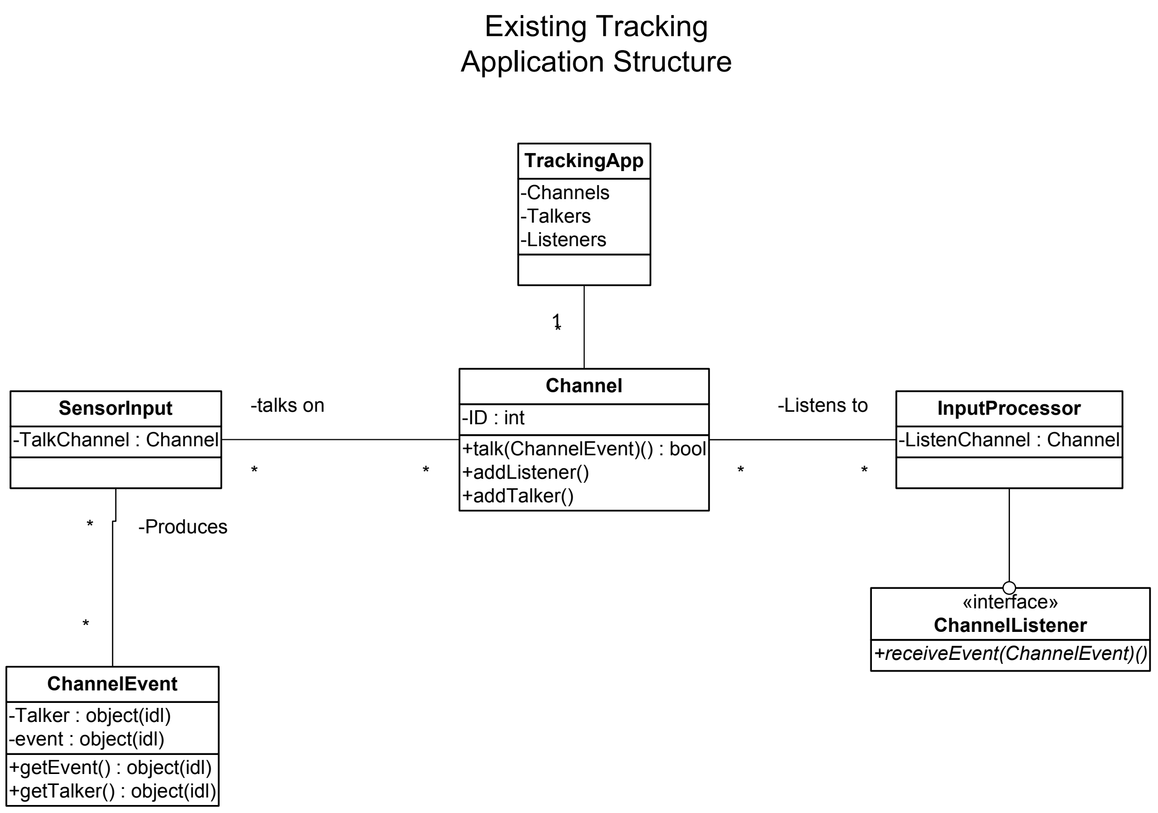
3.6 Assume that a SensorInput object has received a piece of input data. Describe the sequence of method calls that occur to get the data from the SensorInput to the InputProcessor.

TEST 2 PROJECT WRITE-UP

This section contains the project write-up as presented to participants of Test 2.

Contact Tracking Application Project Redesign

For this session, assume that the following Tracking application structure has already been implemented. Recall that the SensorInput class is a Talker that pushes ChannelEvent objects to the Channel by calling the Channel.talk() methods; and, the InputProcessor class implements the ChannelListener interface so that the Channel can deliver events to each InputProcessor by calling their receiveEvent() methods. Note that it was chosen for each Talker and Listener to contain a data member that is a reference to the Channel object that each has registered with; and, each registers with its Channel by calling the Channel's addTalker() or addListener() methods.



Now, for this session you will use many instances of both the SensorInput class and the InputProcessor class. You will have an AirSensorInput, a SurfaceSensorInput, and a SubSurfaceSensorInput; likewise, you will have an AirInputProcessor, a SurfaceInputProcessor, and a SubSurfaceInputProcessor.

By following the existing class structure in the diagram above, the TrackingApp class would create each of the listed instances and would create a Channel object between each pair. For example, a Channel instance named AirChannel would connect the AirSensorInput to the AirInputProcessor. This would require that the TrackingApp keep track of all the Channels. However, the Channel API contains a ChannelManager Class and a ChannelManagerAuthority interface that may be useful in managing the Channels for you.

Below, is a task list that will lead you to investigate both of these API features. As you complete the tasks please verbalize your thought process. Specifically, mention the class, interface, and method names you consider. Also, mention any confusion that arises from the names, methods, and method arguments.

Task1: Redesign the Tracking Application to utilize the ChannelManager class to aid in managing the Channels.

1.1 Based on the ChannelManagerclass name, state the operations you would expect the class to perform?

1.2 Based on the class methods, describe the purpose of the ChannelManager.

1.3 Determine how the choice to use this class will affect the existing program structure.

1.4 Determine how the Channel objects will now be created.

1.5 Determine how the use of this class will affect the structure of the SensorInput and InputProcessor classes.

1.6 Describe how to assemble one pair of SensorInput and InputProcessor objects using the ChannelManager.

Task2: Redesign the Tracking Application to utilize the ChannelManagerAuthority Interface to aid in managing the Channels.

2.1 Based on the interface name, speculate on the functionality you would expect such an object to perform.

2.2 Based on the interface methods, speculate on the purpose for using such an object.

2.3 Speculate on the responsibilities an implementation of this interface would have to fulfill.

2.4 Assume that you will not use a ChannelManager Object at all. Determine how you would use the ChannelManagerAuthority Interface to manage Channels.

2.5 Describe how you would use the ChannelManagerAuthority to assemble one pair of SensorInput and InputProcessor objects.

Task3: Redesign the application to use the ChannelManger class and the ChannelManagerAuthority interface together.

3.1 Discuss how the ChannelManager uses the ChannelManagerAuthority.

3.2 Discuss the similarities between the two names.

3.3 Determine if use of the two objects together changes the implementation of the ChannelManagerAuthority interface. Does it change the role of the CMA?

The Channel API provides a mechanism that allows a Channel to filter events before dispatching them to Listeners. This keeps the Channel from having to deliver every event to every one of its Listeners. Consider the overall Channel Model attributes such as how each object is distinguished by the Channel when registering and when events are delivered to the Channel. Specifically, consider qualities of the Talkers, events, and Listeners that may distinguish them.

Assume you have a Channel object, a SensorInput object, and an InputProcessor object.

Task4: Create a Filter that will allow a Channel to only deliver desired events to the InputProcessor.

4.1 Determine the API feature you will use to create this filter.

4.2 Describe an overview of how this filtering process will work. Include the roles of the SensorInput, Channel, InputProcessor, and Filter objects.

4.3 Discuss possible attributes of the four objects in 4.2 you could use to do the filtering.

4.4 Discuss the method calls that occur when a Channel uses the filter.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

1. Eryigit, Cihat. *A Highly Adaptable Generic Event-Based Message Channel Design for Loosely Coupled Software Modules*, Master's Thesis, Naval Postgraduate School, Monterey, CA, March 2002.

2. McLellan, Samuel G., et al., "Building More Usable APIs," *IEEE* Software, volume: 15 Issue 3, pp. 78-86, May/June 1998.

3. Rubin, Jeffrey, *Handbook of Usability Testing: How To Plan, Design, and Conduct Effective Tests*. John Wiley & Sons, Inc., 1994.

4. Java Foundation Classes: Swing GUI Components. 12 April 2003. Sun Microsystems. 22 May 2003. <http://java.sun.com/products/jfc/#swing>

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, VA

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, CA

3. Geoffrey Xie
   Naval Post Graduate School
   Monterey, CA

4. Rudolph Darken
   Naval Post Graduate School
   Monterey, CA